



PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ  
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA  
DEPARTAMENTO DE ENGENHARIA MECATRÔNICA

*APOSTILA DE TÉCNICAS DE PROGRAMAÇÃO*

CURITIBA  
JANEIRO/2002

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ  
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA  
DEPARTAMENTO DE ENGENHARIA MECATRÔNICA

*APOSTILA DE TÉCNICAS DE PROGRAMAÇÃO*

ELABORAÇÃO: *PROFESSOR MARCELO RUDEK*

COLABORAÇÃO: *GELSON LUIZ CARNEIRO*

*ADRIANO ZELAZOWSKI PEREIRA*

CURITIBA

JANEIRO/2002

<b><u>APOSTILA DE TÉCNICAS DE PROGRAMAÇÃO .....</u></b>	<b><u>0</u></b>
<b><u>APOSTILA DE TÉCNICAS DE PROGRAMAÇÃO .....</u></b>	<b><u>1</u></b>
<b><u>I. CONTRATO DIDÁTICO .....</u></b>	<b><u>8</u></b>
I. INFORMAÇÕES GERAIS .....	8
II. SISTEMA DE AVALIAÇÃO.....	8
III. O PROGRAMA DE APRENDIZAGEM (P.A.).....	9
<b><u>II. APRESENTAÇÃO DO PROGRAMA DE APRENDIZAGEM (P.A.) .....</u></b>	<b><u>10</u></b>
I. CONTEÚDO PROGRAMÁTICO: .....	10
II. BIBLIOGRAFIA RECOMENDADA: .....	10
<b><u>1. INTRODUÇÃO À PROGRAMAÇÃO: ALGORITMOS .....</u></b>	<b><u>11</u></b>
1.1. EXEMPLOS .....	11
1.2. ALGORITMOS EM PORTUGOL .....	12
1.3. PORTUGOL .....	12
1.4. VARIÁVEIS .....	13
1.4.1. DECLARAÇÃO DE VARIÁVEIS.....	13
1.4.1.1. Tipos de Variáveis .....	14
1.4.1.2. Identificadores de Variáveis .....	14
1.4.2. CONSTANTES .....	16
1.5. ESTRUTURA DO ALGORITMO EM PORTUGOL.....	16
1.5.1. COMANDO DE ATRIBUIÇÃO (<-) .....	16
1.5.2. OPERADORES ARITMÉTICOS .....	17
1.5.3. ENTRADA E SAÍDA DE DADOS.....	18
1.5.4. REGRAS PARA ESCREVER ALGORITMOS EM PORTUGOL .....	19
1.5.5. EXERCÍCIOS .....	19
1.6. COMANDOS DE CONTROLE .....	19
1.6.1. DESVIO CONDICIONAL.....	19
1.6.1.1. Operadores Lógicos .....	19
1.6.1.2. Operadores Relacionais .....	20
1.6.1.3. Desvio Condicional Simples.....	20
1.6.1.4. Desvio Condicional Composto .....	21
1.6.2. LAÇOS DE REPETIÇÃO (LOOP).....	23
1.6.2.1. Comando: enquanto/faça .....	24
1.6.2.2. Comando: para / até / faça .....	26
<b><u>2. PROGRAMAÇÃO EM LINGUAGEM C.....</u></b>	<b><u>28</u></b>
2.1. INTRODUÇÃO A PROGRAMAÇÃO EM LINGUAGEM C.....	28

2.1.1. DECLARAÇÃO DE VARIÁVEIS.....	28
2.1.2. COMANDO DE ATRIBUIÇÃO:.....	30
2.1.3. BLOCOS DE COMANDOS:.....	30
<b>2.2. BORLAND C++ BUILDER.....</b>	<b>31</b>
2.2.1. O AMBIENTE DE DESENVOLVIMENTO.....	31
2.2.2. A INTERFACE DE DESENVOLVIMENTO.....	32
3.2.2.1. Barra de Componentes.....	32
2.2.2.2. Formulário (form).....	32
2.2.2.3. Barra de Propriedades.....	33
2.2.3. A CRIAÇÃO DE PROGRAMAS.....	33
A) ENTRADA DE DADOS.....	34
B) ATRIBUIÇÃO.....	35
C) SAÍDA DE DADOS.....	35
E) OPERADORES RELACIONAIS.....	35
2.2.4. PASSOS PARA CRIAR UMA APLICAÇÃO EM C.....	35
a) Abrindo o C++ Builder.....	36
b) Adicionando Formulário.....	36
c) Inserindo Componentes no Formulário.....	36
d) Codificação do Programa.....	37
e) Compilando um Programa.....	38
f) Executando um Programa.....	38
g) Salvando o Programa.....	39
2.2.5. EXERCÍCIOS.....	39
<b>2.3. ESCOPO DE VARIÁVEIS.....</b>	<b>40</b>
2.3.1. VARIÁVEIS LOCAIS.....	40
EXEMPLO.....	40
2.3.2. VARIÁVEIS GLOBAIS.....	41
<b>2.4. DESVIO CONDICIONAL EM C.....</b>	<b>42</b>
2.4.1. DESVIO CONDICIONAL SIMPLES.....	42
2.4.2. DESVIO CONDICIONAL COMPOSTO.....	42
2.4.3. IF'S ANINHADOS.....	43
2.4.4. EXEMPLO.....	43
SOLUÇÃO SEM O USO DE “IF”.....	43
2.4.5. EXERCÍCIO.....	45
<b>2.5. LAÇOS DE REPETIÇÃO EM C.....</b>	<b>46</b>
2.5.1. LOOP PARA/FAÇA (FOR).....	46
2.5.2. LOOP ENQUANTO/FAÇA (WHILE).....	46
2.5.3. LOOP FAÇA/ENQUANTO (DO/WHILE).....	47
2.5.4. EXEMPLO.....	47
2.5.5 EXERCÍCIOS.....	48
<b>2.6. PROCEDIMENTOS EM C.....</b>	<b>49</b>
2.6.1. DEFINIÇÃO.....	49
2.6.2. EXEMPLO 1.....	49
2.6.3. PROTÓTIPO.....	52
2.6.4. PARÂMETROS.....	53
2.6.5. EXEMPLO 2.....	53
<b>2.7. FUNÇÃO EM C.....</b>	<b>56</b>
2.7.1. DEFINIÇÃO.....	56

2.7.2. DECLARAÇÃO.....	56
2.7.3 PARÂMETROS E RETORNO .....	56
2.7.4. EXEMPLO 1 .....	57
2.7.5. EXEMPLO 2 .....	59
2.7.6. EXERCÍCIOS .....	61
<b>2.8. INCREMENTOS E DECREMENTOS .....</b>	<b>62</b>
2.8.1. INCREMENTO/DECREMENTO A POSTERIORI .....	62
2.8.2. INCREMENTO/DECREMENTO A PRIORI .....	63
2.8.3. EXERCÍCIO .....	63
<b>2.9. ATRIBUIÇÃO COMPOSTA .....</b>	<b>64</b>
2.9.1. EXERCÍCIO .....	64
<b>2.10. ATRIBUIÇÃO MÚLTIPLA.....</b>	<b>64</b>
2.10.1. EXEMPLO .....	64
<b>2.11. OPERADOR INTERROGAÇÃO (?) .....</b>	<b>65</b>
<b>2.12. NÚMEROS ALEATÓRIOS .....</b>	<b>65</b>
2.12.1. SINTAXE DO COMANDO .....	65
2.12.2. EXEMPLO .....	65
<b>2.13 COMANDO SWITCH/CASE.....</b>	<b>66</b>
2.13.1. SINTAXE DO COMANDO.....	66
2.13.2. EXEMPLO .....	66
<b>2.14. TIMER .....</b>	<b>67</b>
2.14.1. O COMPONENTE TIMER NO C++ BUILDER.....	67
2.14.2. AS PROPRIEDADES DO TIMER .....	68
2.14.3. EXEMPLO .....	68
2.14.4. EXERCÍCIO.....	68

**3. ESTRUTURAS HOMOGÊNEAS DE DADOS .....** **69**

<b>3.1. MATRIZES UNIDIMENSIONAIS (VETORES).....</b>	<b>69</b>
3.1.1. EXEMPLOS .....	69
3.1.2. INDEXAÇÃO.....	69
3.1.3. EXEMPLO .....	70
3.1.4. EXERCÍCIO.....	71
<b>3.2. ORDENAÇÃO DE VETORES .....</b>	<b>71</b>
3.2.1. ALGORITMO DE ORDENAÇÃO (BOLHA).....	71
3.2.2. EXERCÍCIO.....	72
<b>3.3. STRINGS.....</b>	<b>73</b>
3.3.1. EXEMPLO 1 .....	73
3.3.2. EXEMPLO 2 .....	74
3.3.3. COPIANDO STRINGS .....	74
3.3.4. COMPARAÇÃO DE STRINGS .....	75
3.3.5. TAMANHO DE STRINGS .....	75
3.3.6. COMPARAÇÃO DE ELEMENTOS DA STRING.....	76
3.3.7. CONVERSÃO DE TIPOS .....	76
3.3.7.1. convertendo valores numéricos para caracter .....	77
3.3.7.2. convertendo string para valores numéricos .....	77
3.3.8 EXERCÍCIOS .....	78
<b>3.4. MATRIZES .....</b>	<b>79</b>

3.4.1. MATRIZES BIDIMENSIONAIS .....	79
3.4.2. MATRIZES MULTIDIMENSIONAIS .....	80
3.4.3. MATRIZES DE STRINGS .....	80
3.4.4. EXERCÍCIOS .....	82
9) DESENVOLVA UM PROGRAMA QUE PERMITA MOVIMENTAR O ELEMENTO 1 DA MATRIZ ABAIXO EM UMA DIREÇÃO ALEATÓRIA A CADA 1S. O MOVIMENTO DO ELEMENTO NÃO PODE EXTRAPOLAR OS LIMITES DA MATRIZ.....	82

**4. PONTEIROS EM C .....83**

<b>4.1. DEFINIÇÃO .....</b>	<b>83</b>
<b>4.2. DECLARAÇÃO DE UM PONTEIRO .....</b>	<b>84</b>
<b>4.3. EXEMPLOS .....</b>	<b>85</b>
<b>4.4. PONTEIROS PARA MATRIZ .....</b>	<b>87</b>
<b>4.5. VETORES DE PONTEIROS .....</b>	<b>89</b>
4.5.1. EXEMPLO 1 .....	89
4.5.2. EXERCÍCIO.....	91
4.5.3. EXEMPLO 2 .....	91
4.5.4. EXERCÍCIOS .....	92

**5. ALOCAÇÃO DINÂMICA DE MEMÓRIA .....93**

<b>5.1. INTRODUÇÃO .....</b>	<b>93</b>
<b>5.2. COMANDO DE ALOCAÇÃO .....</b>	<b>93</b>
5.2.1. EXEMPLO DE ALOCAÇÃO USANDO O COMANDO MALLOC().....	93
5.2.2. MELHORANDO O USO DE PONTEIROS.....	96
<b>5.3. EXERCÍCIOS.....</b>	<b>97</b>
<b>5.4. PORTABILIDADE.....</b>	<b>97</b>
5.4.1. EXEMPLO DO USO DE SIZEOF .....	98
<b>5.5. EXERCÍCIOS .....</b>	<b>98</b>

**6. ARQUIVOS EM C .....103**

<b>6.1. PONTEIRO DE ARQUIVO.....</b>	<b>103</b>
<b>6.2. ABRINDO ARQUIVOS.....</b>	<b>103</b>
6.2.1. ARQUIVOS TIPO TEXTO .....	104
6.2.2. ARQUIVOS BINÁRIOS .....	105
<b>6.3. ABRINDO UM ARQUIVO PARA ESCRITA .....</b>	<b>105</b>
6.3.1. OBSERVAÇÕES .....	106
<b>6.4. ABRINDO UM ARQUIVO PARA LEITURA .....</b>	<b>107</b>
<b>6.5. FECHANDO UM ARQUIVO.....</b>	<b>107</b>
<b>6.6. COMANDOS DE ESCRITA E LEITURA.....</b>	<b>108</b>
6.6.1. FPUTC().....	108
6.6.2. FGETC().....	110
6.6.3. EXERCÍCIO COM FPUTC() E FGETC() .....	111
<b>6.7. GRAVAÇÃO DE STRINGS COM FPUTS() .....</b>	<b>111</b>

<b>6.8. LEITURA DE STRINGS COM FGETS()</b> .....	<b>112</b>
<b>6.9. EXERCÍCIOS COM FPUTS() E FGETS()</b> .....	<b>113</b>
<b>6.10. LEITURA COM FREAD()</b> .....	<b>113</b>
<b>6.11. GRAVAÇÃO COM FWRITE()</b> .....	<b>114</b>
<b>6.12. GRAVAÇÃO COM FPRINTF()</b> .....	<b>115</b>
<b>6.13. LEITURA COM FSCANF()</b> .....	<b>115</b>
<b>6.14. EXERCÍCIOS</b> .....	<b>116</b>
<b><u>7. REGISTROS</u></b> .....	<b><u>117</u></b>
<b>7.1. DEFINIÇÃO</b> .....	<b>117</b>
<b>7.2. INICIALIZAÇÃO</b> .....	<b>117</b>
7.2.1. EXEMPLO 1 .....	118
7.2.2. EXEMPLO 2 .....	118
<b>7.3. ACESSO AOS ELEMENTOS DA ESTRUTURA</b> .....	<b>118</b>
<b>7.4. EXERCÍCIO</b> .....	<b>119</b>
<b>7.5. MATRIZES DE ESTRUTURAS</b> .....	<b>119</b>
7.5.1. EXEMPLO .....	119
7.5.2. EXERCÍCIO.....	120
<b>7.6. USO DE TYPEDEF</b> .....	<b>122</b>
7.6.1. EXEMPLO .....	122
7.6.2. EXEMPLO 2 .....	122
<b>7.7. GRAVAÇÃO E LEITURA DE REGISTROS</b> .....	<b>123</b>
7.7.1 EXEMPLO .....	123
7.7.2. EXERCÍCIO.....	123
<b>7.8. PONTEIROS PARA REGISTROS</b> .....	<b>124</b>
7.8.1. EXEMPLO .....	124
<b><u>8. GRÁFICOS EM C</u></b> .....	<b><u>127</u></b>
<b>8.1. INTRODUÇÃO</b> .....	<b>127</b>
<b>8.2. DESENHANDO LINHAS</b> .....	<b>127</b>
<b>8.3. USANDO O PAINTBOX</b> .....	<b>129</b>
<b>8.4. COMPONENTE PANEL</b> .....	<b>130</b>
<b>8.5. DESENHANDO RETÂNGULOS</b> .....	<b>131</b>
<b>8.6. DESENHANDO ELIPSES</b> .....	<b>132</b>
<b>8.7. DESENHANDO PONTOS (PIXELS)</b> .....	<b>133</b>
<b>8.8. EXEMPLO</b> .....	<b>133</b>
<b>8.9 EXERCÍCIOS</b> .....	<b>137</b>
<b><u>9. LISTAS LINEARES</u></b> .....	<b><u>138</u></b>
<b>9.1. FILA</b> .....	<b>138</b>
9.1.1. DEFINIÇÃO.....	138
9.1.2. OBJETIVO.....	138
9.1.3. EXEMPLO .....	138

<b>9.2. FILA CIRCULAR</b> .....	<b>141</b>
<b>9.3. PILHA</b> .....	<b>143</b>
9.3.1. DEFINIÇÃO.....	143
9.3.2. EXEMPLO .....	143
<b>9.4. EXERCÍCIOS</b> .....	<b>144</b>
<b>9.5. LISTAS ENCADEADAS</b> .....	<b>145</b>
<b>9.6. EXEMPLO</b> .....	<b>146</b>
<b>9.7. EXERCÍCIO</b> .....	<b>147</b>
<b>9.8. EXEMPLO</b> .....	<b>147</b>
<b>9.9. OPERAÇÕES COM LISTA ENCADEADA</b> .....	<b>148</b>
<b>9.10. EXEMPLO</b> .....	<b>149</b>
<b>9.11. LISTAS DUPLAMENTE ENCADEADAS</b> .....	<b>151</b>
<b>9.12. EXEMPLO</b> .....	<b>151</b>
<b>9.13. EXEMPLO</b> .....	<b>152</b>
<b><u>10. RECURSIVIDADE</u></b> .....	<b><u>155</u></b>
<b>10.1. INTRODUÇÃO</b> .....	<b>155</b>
<b>10.2. EXEMPLO</b> .....	<b>155</b>
<b>10.3. EXERCÍCIOS</b> .....	<b>156</b>
<b><u>11. EXERCÍCIOS COM VETORES</u></b> .....	<b><u>157</u></b>
<b><u>12 -EXERCÍCIOS COM MATRIZES</u></b> .....	<b><u>158</u></b>
<b><u>13. EVENTOS DE FORMULÁRIO E VARIÁVIES EXTERNAS</u></b> .....	<b><u>160</u></b>
<b>13.1. EXERCÍCIO PROPOSTO</b> .....	<b>160</b>
<b>13.2. LISTAGEM DO PROGRAMA</b> .....	<b>161</b>
13.2.1. UMEDIA1.CPP .....	161
13.2.2. UMEDIA.CPP .....	162
<b><u>14. ROTINAS DE ORDENAÇÃO</u></b> .....	<b><u>164</u></b>
<b><u>15. COMPONENTES DO C++ BUILDER E SUAS PRINCIPAIS PROPRIEDADES.</u></b> ..	<b><u>167</u></b>
<b>15.1. BITBTN</b> .....	<b>167</b>
15.1.1. PRINCIPAIS PROPRIEDADES .....	167
15.1.2. EXEMPLO .....	168
<b>15.2 CHECKBOX</b> .....	<b>168</b>
15.2.1. PRINCIPAIS PROPRIEDADES .....	168
15.2.2. EXEMPLO .....	169
<b>15.3. COMBOBOX</b> .....	<b>169</b>
15.3.1. PRINCIPAIS PROPRIEDADES .....	169
15.3.2. EXEMPLO .....	170



<b>15.4. LISTBOX.....</b>	<b>171</b>
15.4.1. PRINCIPAIS PROPRIEDADES .....	171
15.4.2. EXEMPLO .....	171
<b>15.5. PAGECONTROL .....</b>	<b>172</b>
15.5.1. PRINCIPAIS COMANDOS.....	173
15.5.2. EXEMPLO .....	173
<b>15.6. RADIOBUTTON .....</b>	<b>174</b>
15.6.1. PRINCIPAIS PROPRIEDADES .....	174
15.6.2. EXEMPLO .....	174
<b>15.7. RADIOGROUP.....</b>	<b>175</b>
15.7.1. PRINCIPAIS PROPRIEDADES .....	175
15.7.2. EXEMPLO .....	175
<b>15.8. SCROLLBAR .....</b>	<b>176</b>
15.8.1. PRINCIPAIS PROPRIEDADES .....	177
15.8.2. EXEMPLO .....	177
<b>15.9. SPEEDBUTTON.....</b>	<b>178</b>
15.9.1. PRINCIPAIS PROPRIEDADES .....	178
15.9.2. EXEMPLO .....	178
<b>15.10. STRINGGRID.....</b>	<b>179</b>
15.10.1. PRINCIPAIS PROPRIEDADES .....	179
15.10.2. EXEMPLO .....	180
<b>15.11. TABCONTROL.....</b>	<b>180</b>
15.11.1. PRINCIPAIS PROPRIEDADES .....	180
15.11.2. EXEMPLO .....	181

## I. Contrato Didático

---

### i. INFORMAÇÕES GERAIS

- Todas as aulas (teoria e prática) serão em laboratório; então todos deverão estar familiarizados com o manuseio dos computadores, pois o conteúdo da aula será fornecido pela internet através do endereço <http://www.las.pucpr.br/rudek>
- Trazer disquetes para armazenar os trabalhos de aula;
- Quem necessitar pode (e deve) usar os laboratórios da PUC para aprender a usar o computador; cada laboratório possui um monitor responsável que pode ajudar nas dúvidas, fora dos horários de aula;
- Observar as normas de utilização do laboratório e das aulas de TP1, principalmente em relação ao zelo pelos equipamentos e atenção às atividades de aula;
- Informações sobre o curso de Engenharia de Controle e Automação (ECA) e de atividades do Laboratório de Automação e Sistemas (LAS) podem ser obtidas diretamente no site do LAS através do endereço [www.las.pucpr.br](http://www.las.pucpr.br).

### ii. SISTEMA DE AVALIAÇÃO

- A avaliação é contínua durante todo o curso. Será composta de provas, trabalhos, exercícios em sala (participação);
- A nota semestral será composta de 3 notas parciais:, sendo:
  - 1.<sup>a</sup> parcial : prova (80%) + exercícios (20%);
  - 2.<sup>a</sup> parcial: prova (60%) + exercícios (10%) + trabalhos (30 %);
  - 3.<sup>a</sup> parcial: prova (50%) + exercícios (20%) + trabalhos (30 %);

OBS: a forma de avaliação das parciais, pode variar de acordo com o andamento das aulas e nível de aprendizado da turma.
- Os trabalhos serão divulgados na minha página; Trabalhos entregues com atraso não serão considerados, ou terão nota reduzida (2,0 pts por dia de atraso), de acordo com critério do professor.
- Faltas: no máximo 12 faltas (4 dias de aula). Mais de 12 faltas o aluno estará automaticamente reprovado por falta, independente das notas que possua.

**iii. O PROGRAMA DE APRENDIZAGEM (P.A.)**

Este programa de aprendizagem explora o estudo de algoritmos e programação em linguagem C. Para aprovação neste semestre o aluno deverá estar apto a raciocinar e desenvolver a sua capacidade de abstração, para a criação de algoritmos e programas elementares.

As técnicas de programação serão vistas através da utilização de uma pseudolinguagem em português, denominada "Portugol". Assim, o aluno poderá escrever algoritmos e aplicar as técnicas de desenvolvimento para serem usadas com qualquer linguagem de programação. Para este P.A., será utilizada a linguagem C padrão, em ambiente Windows.

Na seção seguinte será apresentado o conteúdo programático de Técnicas de Programação I e II (TPI e TPII).

## *II. Apresentação do Programa de Aprendizagem (P.A.)*

---

### **i. CONTEÚDO PROGRAMÁTICO:**

- Conceitos de Informática e Programação;
  - . Princípios de funcionamento;
  - . Sistemas Operacionais (DOS<sup>®</sup> e WINDOWS<sup>®</sup>);
  - . Sistemas Numéricos;
  - . Memória e Dispositivos de Armazenamento;
  - . Linguagens de Programação;
- Algoritmos: Introdução;
- Algoritmos: Portugal;
- Introdução a Linguagem C : Principais comandos e programação para Windows;
  
- OBS: Conteúdo das aulas em [www.las.pucpr.br/rudek](http://www.las.pucpr.br/rudek)

### **ii. Bibliografia Recomendada:**

1. Algoritmos e Estruturas de Dados; Guimarães e Lages; LTC – Livros Técnicos e Científicos.
2. Estruturas de Dados Usando C; A. Tanenbaum; Makron Books
3. Dominando Algoritmos com C; Kyle Loundon; Ed. Ciência Moderna.
4. C Completo e Total; Herbert Schildt; Makron Books;
5. C A Linguagem de Programação Padrão Ansi; Brian W. Kernighan, Dennis M. Ritchie; Editora Campus;
6. C++ Builder 5 - Guia Prático; César A Mateus. Ed. Érica ([www.ERICA.com.br](http://www.ERICA.com.br));
7. C++ Builder 5 Developer's Guide; J. Hollinworth, D. Butterfield, et al. Ed. SAMS;
8. C Completo e Total; Herbert Schildt; Makron Books;
9. Internet.

## 1. INTRODUÇÃO À PROGRAMAÇÃO: ALGORITMOS

---

Várias definições de algoritmos estão presentes na literatura (ver bibliografia indicada). De forma geral um algoritmo pode ser definido como:

*Um algoritmo representa de forma estruturada, um padrão de comportamento de eventos ou sequência de ações, que levam a um resultado esperado.*

Resumindo:

algoritmo = como definir o problema, esquematizar, exercício do raciocínio;

técnicas de programação = como operacionalizar, recursos, exercício da implementação.

### 1.1. Exemplos

a) Seqüência de ações para chegar ao trabalho/universidade:

Acordar → levantar → tomar café → pegar o ônibus

Ou → chegar ao destino

→ pegar o carro

Note que, para cada ação acontecer, é necessário que a ação imediatamente anterior tenha sido executada.

Note também que, cada ação pode conter outros eventos associados (outros algoritmos).

b) Manuais de montagem e utilização de equipamentos;

c) Qual o padrão de comportamento utilizado para gerar a sequência abaixo?

1, 5, 9, 13, 17, 21, 25 ...

resposta: \_\_\_\_\_

## **1.2. ALGORITMOS EM PORTUGOL**

Como no item 1 "... um algoritmo é de forma geral, uma descrição passo a passo de como um problema pode ser solucionado. A descrição deve ser finita, e os passos devem ser bem definidos sem ambiguidades" [Terada] . A razão da existência do algoritmo vem da dissonância entre um estado desejado e aquele observado na realidade. Algoritmo não é a solução de um problema, mas é o meio de obtê-la. A resolução de um problema envolve vários parâmetros que devem ser organizados através de alguma técnica formal.

As técnicas de desenvolvimento estruturado de algoritmos, tem o objetivo de:

- Facilitar o desenvolvimento de algoritmos;
- Facilitar o seu entendimento pelos operadores;
- Antecipar a correção;
- Facilitar manutenção e modificações;
- Permitir que o desenvolvimento seja feita por uma equipe de pessoas.

Uma técnica formal afasta a possibilidade de uma ambiguidade. Ou seja, a partir de dadas condições iniciais a execução do algoritmo será realizada por um mesmo "caminho" (sequência de ações), que deve resultar num mesmo estado final. Uma destas técnicas é o portugol.

## **1.3. PORTUGOL**

Portugol é uma pseudolinguagem que permite ao programador pensar no problema em si e não no equipamento que irá executar o algoritmo. Devem ser considerados a sintaxe (em relação à forma) e a semântica (em relação ao conteúdo ou seu significado). Em portugol a sintaxe é definida pela linguagem e a semântica depende do significado que quer se dar ao algoritmo.

No portugol e nas linguagens de programação, basicamente têm-se comandos e variáveis que operacionalizam a execução de um algoritmo. Estes comandos são

executados sequencialmente, de forma que um comando só será executado após a finalização do comando anterior.

A estrutura de um algoritmo em portugol pode ser dada como:

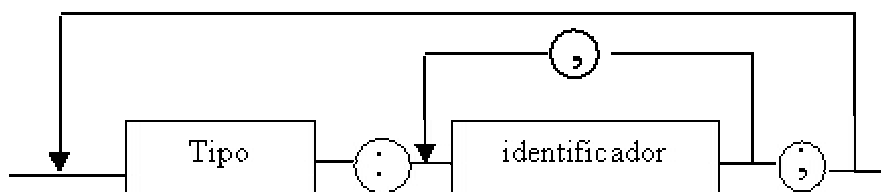
**Exemplo:**

início  
 <declarações de variáveis>  
 <comandos>  
fim

**1.4. Variáveis**

**1.4.1. Declaração de Variáveis**

Uma variável é um local (área na memória do computador) que armazena um tipo específico de conteúdo. Uma variável contém um valor que se modifica durante a execução do programa. A variável possui um identificador (nome), que pode ser representado da seguinte forma:



### 1.4.1.1. Tipos de Variáveis

Variáveis são componentes das linguagens de programação, que identificam os valores que estão sendo manipulados pelos programas. Uma variável, como o próprio nome sugere, contém valores que variam de acordo com a execução do programa. Uma variável deve possuir um tipo específico. As variáveis em português, são divididas em 4 tipos principais, (embora na linguagem C existam modificações para estes tipos principais).

No português, os tipos básicos de variáveis são:

- **Inteiro:** Qualquer número inteiro (negativo, nulo ou positivo).  
Exemplo: -100, 0, 1, 2, 1250.
- **Real:** Qualquer número real, nulo ou positivo.  
Exemplo: -10, -1.5, 11.2, 0,1, 2, 50.
- **Caracter:** Caracteres alfanuméricos.  
Exemplo: casa, Win31, 123, alfa#2, etc...
- **Lógico:** valor lógico verdadeiro ou falso  
Exemplo:  $x > y$  ?

Exemplos:     inteiro: valor; // a variável valor é do tipo inteiro  
                   real: media; // a variável media é do tipo real  
                   caracter: nome\_aluno; // a variável nome\_aluno é do tipo caracter  
                   lógico: maior; // a variável maior é do tipo booleano

### 1.4.1.2. Identificadores de Variáveis

O identificador de uma variável, se refere ao nome de como ela vai ser conhecida no programa. É importante não esquecer que:

- a) Não é possível definir variáveis de diferentes tipos com o mesmo identificador (nome); O exemplo: **real** A; **inteiro** A; causaria erro na programação, mas pode ser



usado **real** A1; **inteiro** A2; ou normalmente um nome mais significativo, como **real** media, **inteiro** valor, **caracter** nome, etc.

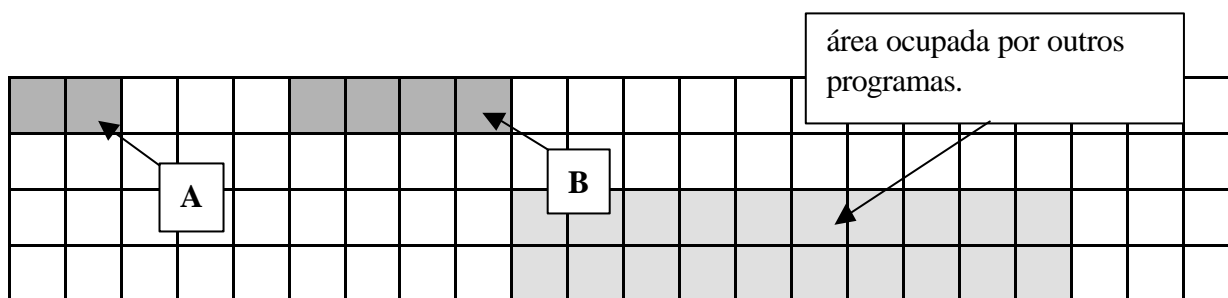
- b) Tomar alguns cuidados em relação à sintaxe da linguagem, por exemplo, não é possível ter identificador como: **caracter** ?nome, **real** valor\*, **inteiro** 1x, .
- c) .Letras maiúsculas e minúsculas são tratadas de forma diferente, então **Media** é diferente de **media**, como também de **MEDIA**.

Cada variável definida no programa usa um local da memória, que é acessada através do nome dado a variável. O espaço de memória ocupado pelo conteúdo da variável, depende do tamanho destes tipos de dados, que variam de acordo com o tipo do processador e com a implementação do compilador. Como referência inicial para este estudo sobre variáveis, pode-se considerar pelo ANSI C, o seguinte:

- Tipo Inteiro com 2 bytes;
- Tipo real com 4 bytes;
- Tipo caracter com 1 byte;

Exemplo:

Pode-se supor a memória como uma matriz, como a figura abaixo, onde cada célula possui tamanho de 1 byte (8 bits):



Para armazenar o valor inteiro A= 1, necessita-se de 2 bytes (1 inteiro = 2 bytes na memória \*);  
 Para armazenar o valor real B= 1, necessita-se de 4 bytes (1 real = 4 bytes na memória \*);

\* no C ANSI; no C++ Builder os tamanhos das variáveis são diferentes.

### 1.4.2 Constantes

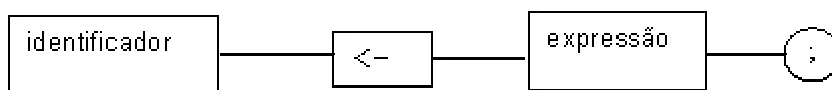
Uma constante é um valor fixo, que não se modifica ao longo do tempo, durante a execução do programa. Em algoritmos representaremos constantes pelo tipo const, constante ou #define (eventualmente, na elaboração dos algoritmos, alguns elementos da linguagem C podem ser escritos no algoritmo).

Exemplo:        **const** M 10;

### 1.5. Estrutura do Algoritmo em Portugol

#### 1.5.1. Comando de Atribuição (<-)

A sintaxe do comando é dada por:



Exemplos:

a) atribuição de um valor constante	<b><u>inteiro</u></b> valor; valor <- 10;
b) atribuição entre variáveis	<b><u>inteiro</u></b> valor; <b><u>inteiro</u></b> x; x <- 10; valor <- x;
c) resultado de expressões:	<b><u>inteiro</u></b> valor; <b><u>inteiro</u></b> x, y; x <- 10; y <- 5; valor <- x + y * 2;

### 1.5.2. Operadores Aritméticos

Os símbolos das operações básicas são:

- A multiplicação é dada através do operador **\*** (asterisco);

Exemplo: `z <- x * y;`

- A soma é realizada através do operador **+**;

Exemplo: `z <- x + y;`

- A subtração é dada através do operador **-**;

Exemplo: `z <- x - y;`

- A divisão para real será dada por **/**;

Exemplo: `z <- x / y;`

- A divisão para inteiro será dada por **div**;

Exemplo: `z <- x div y;`

- O resto de uma divisão é dada pelo comando **mod**.

Exemplo: `z <- x mod y;`

- O cálculo de  $x^y$  é dado pelo símbolo **^**.

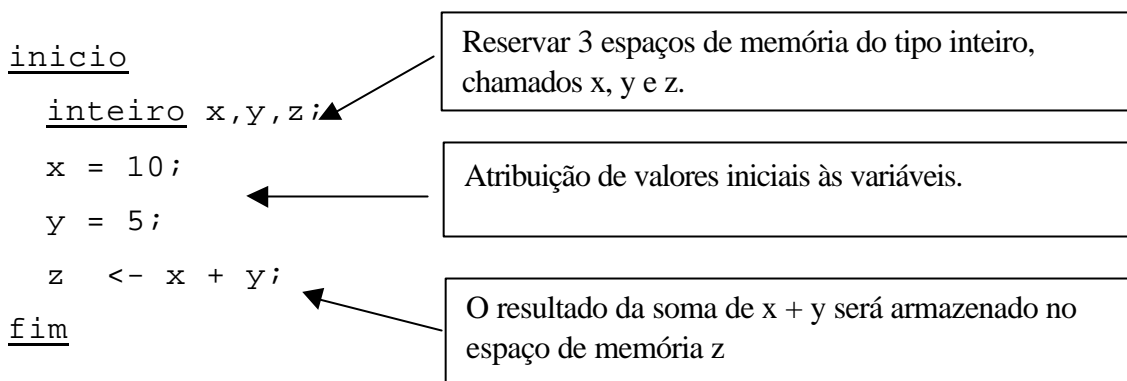
Exemplo: `z <- x^y;`

- A raiz de uma valor é extraída através do comando **raiz()**.

Exemplo: `z <- raiz(x);`

Exemplos

- a) Desenvolva um algoritmo em portugol para somar dois valores inteiros (10 + 5)



Obs:

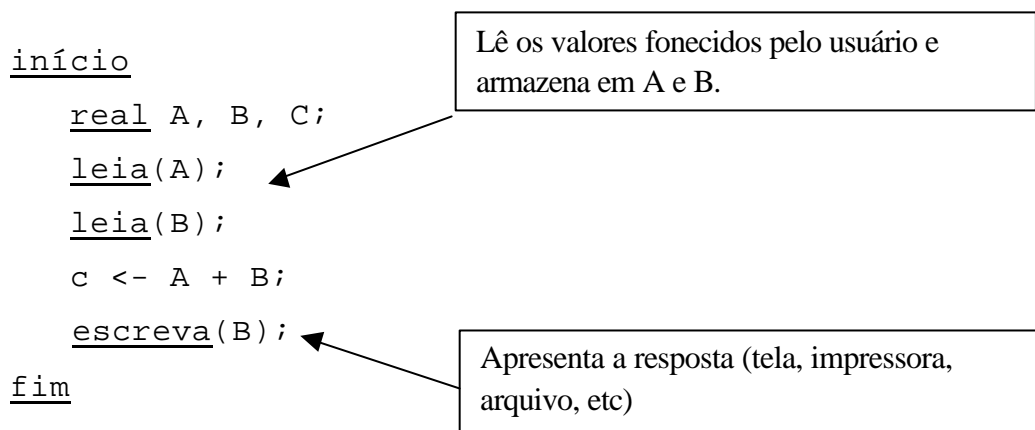
- O inconveniente deste algoritmo é que sempre fornecerá o mesmo resultado, o que não é interessante. De maneira mais correta, os valores de  $x$  e  $y$  podem ser fornecidos pelo usuário, permitindo ao algoritmo que efetue a soma de dois números quaisquer.
- Neste exemplo podemos observar que o resultado da soma de  $x + y$  será armazenado em  $z$ . Como o usuário ficará sabendo da resposta ? É necessário usar comandos de escrita, para apresentar os resultados.

### 1.5.3. Entrada e Saída de Dados

Na construção de algoritmos, é conveniente que o usuário possa informar dados externos, para serem operados pelo programa. Assim, um programa pode receber um dado informado por um operador através de um comando de leitura. Da mesma forma, pode ser necessário conhecer o resultado de determinada operação executada pelo computador, então será necessária uma forma de exibir os dados.

Cada linguagem tem uma forma específica para entrada e saída de dados. Em algoritmos usaremos os comandos genéricos leia() e escreva(), para realizar a interface com o usuário.

#### **Exemplo:**



### **1.5.4. Regras para escrever algoritmos em portugol**

- Incluir comentários pelo menos nas linhas mais importantes do programa;
- Usar nomes significativos para as variáveis e constantes, que possam identificar o conteúdo;
- Grifar as palavras chaves do portugol;
- Alinhar os comandos facilita a legibilidade do algoritmo e reduz a possibilidade de erros.

### **1.5.5. Exercícios**

- 1 – Desenvolva um algoritmo em portugol para calcular  $x^y$ . Os valores de x e y serão fornecidos pelo usuário do programa;
- 2 – Desenvolva um programa que calcule o volume de uma esfera de raio R, fornecido pelo usuário. [  $V = \frac{4}{3} \pi R^3$  ]
- 3 – Desenvolva um programa que transforme um valor de temperatura fornecido pelo usuário, de Fahrenheit ( F ) para Graus Celcius ( °C ). [  $V = \frac{5}{9} (F - 32)$  ]
- 4 – Desenvolva um algoritmo para calcular a média entre 4 valores fornecidos pelo usuário.
- 5 – Desenvolva um algoritmo para encontrar as raízes de uma equação do tipo  $Ax^2 + Bx + C$ .

## **1.6. Comandos de Controle**

Os comandos de controle permitem alterar a direção tomada por um programa (desvio), ou fazer com que partes específicas de um algoritmo seja executada mais de uma vez (loop).

### **1.6.1. Desvio Condicional**

Muitas vezes será necessário desviar a execução do programa segundo uma condição. (Exemplo: ir a universidade de carro ou de ônibus ?). Para se testar condições é necessário utilizar operadores lógicos e relacionais.

#### **1.6.1.1. Operadores Lógicos**

Os operadores "e", "ou" e "não" permitem realizar a combinação lógica de variáveis do tipo booleana (lógico).

Para isto utilizam-se as tabelas verdade:

Var1	Var2	E
V	V	V
V	F	F
F	V	F
F	F	F

Var1	Var2	OU
V	V	V
V	F	V
F	V	V
F	F	F

Var1	Não
V	F
F	V

### 1.6.1.2. Operadores Relacionais

Permitem realizar a comparação de conteúdos das variáveis:

A igualdade é dada por	= ;	Maior ou igual, pelo símbolo	>=;
A desigualdade é dada por	<> ;	Menor ou igual, pelo símbolo	<=;
Maior que, pelo símbolo	>;	Não !;	
Menor que, pelo símbolo	<;		

### 1.6.1.3. Desvio Condicional Simples

Para que a execução de um algoritmo seja desviada para uma outra ação, é necessário um comando de desvio. Este comando é dado pelas palavras reservadas **se** e **fim se**. Dentro deste bloco podemos ter vários comandos de atribuição, operações lógicas e aritméticas, e também novos blocos de desvio condicional.

**se** (condição) **então**

lista de comandos...

**fim se**

**Desvio Condicional Simples**

```

início
    inteiro A, B;
    A <- 100;
    B <- 20;
    se A > B então
        A <- B;
        B <- 0;
    fim se
fim

```

**Exercícios**

- 1 – Desenvolva um algoritmo em portugol para calcular  $x^y$ . Os valores de x e y serão fornecidos pelo usuário do programa, e o maior valor deve estar em x e o menor em y;
- 2 – Desenvolva um programa que calcule o volume de uma esfera de raio R, fornecido pelo usuário. Observe que R não pode ser menor que 0 (zero). [  $V = \frac{4}{3} \pi R^3$  ].
- 3 – Desenvolva um programa que transforme um valor de temperatura fornecido pelo usuário, de Fahrenheit ( F ) para Graus Celcius ( °C ), ou Graus Celcius para Fahrenheit, de acordo com uma opção fornecida pelo usuário. [  $V = \frac{5}{9} (F - 32)$  ]
- 4 – Desenvolva um algoritmo para encontrar as raízes de uma equação do tipo  $Ax^2 + Bx + C$ . Observe que o valor de A não pode ser 0 (zero) e o valor do delta não pode ser menor que 0.

**1.6.1.4. Desvio Condicional Composto**

Neste caso as condições, verdadeiro ou falso, podem gerar ações através de um único comando de desvio condicional, adicionando-se o operador **senão** na estrutura condicional, como apresentado abaixo:

```

se (condição) então
    lista de comandos...
    senão
    lista de comandos...
fim se

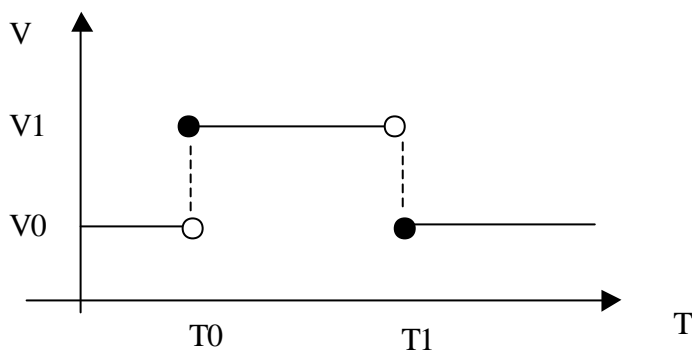
```

```

Desvio Condicional Composto
...
se A > B então
    A <- B;
    B <- 0;
senão
    B <- A;
    A <- 1;
fim se
...
    
```

**Exercícios**

- 1 – Desenvolva um algoritmo que apresente como resposta se um valor inteiro fornecido pelo usuário é par ou ímpar;
- 2 – Dado o gráfico abaixo, testar se um valor T qualquer fornecido pelo usuário, pertence ao intervalo  $T_0 \leq T \leq T_1$ . Dados  $T_0 = 5$ ,  $T_1 = 10$ ,  $V_0 = 1$ ,  $V_1 = 2$ ;



- 3 – Dado o gráfico do exercício 2, desenvolva um algoritmo que apresente o valor de V para um dado valor de T, fornecido pelo usuário.
- 4 – Modifique os exercícios 3 e 4 do item 2.5.1.3. para utilizar um desvio condicional composto, se for o caso.



**1.6.2. Laços de Repetição (loop)**

Uma sequência de ações é repetida por um número específico de vezes, até que uma condição seja satisfeita. Enquanto a condição for verdadeira, as instruções serão executadas. O laço de repetição também pode ser chamado de *loop*.

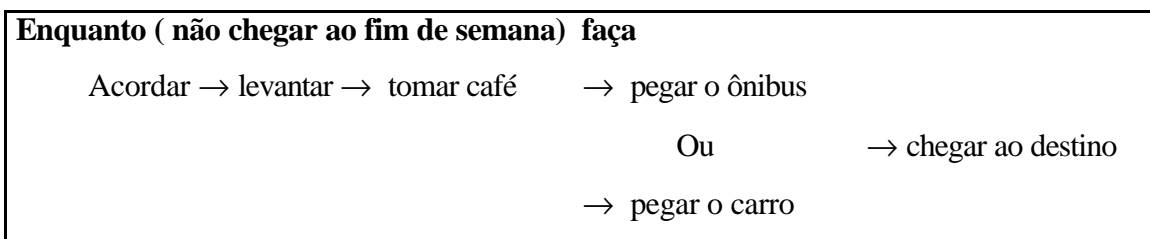
**Exemplo 1:**

Durante uma semana, um mês, etc, vc pode realizar a mesma seqüência de ações, como no exemplo:

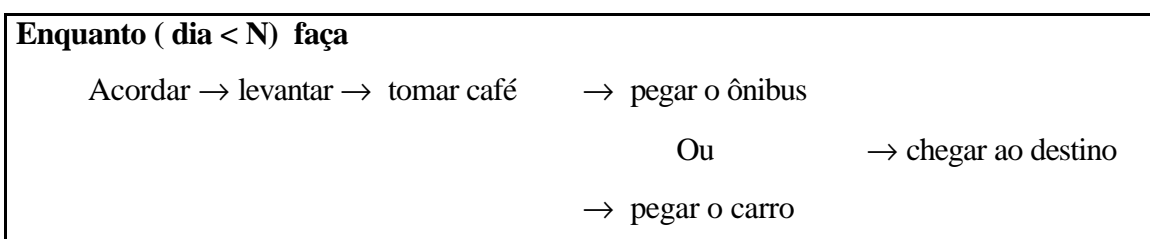
<b>1 ° Dia</b>		
Acordar → levantar → tomar café	→ pegar o ônibus	
	Ou	→ chegar ao destino
	→ pegar o carro	
<b>2 ° Dia</b>		
Acordar → levantar → tomar café	→ pegar o ônibus	
	Ou	→ chegar ao destino
	→ pegar o carro	
•		
•		
•		
<b>N – ésimo Dia</b>		
Acordar → levantar → tomar café	→ pegar o ônibus	
	Ou	→ chegar ao destino
	→ pegar o carro	

Como as ações se repetem durante um período ou até que um evento ocorra (chegar ao fim de semana) , pode-se melhorar escrita da seqüência do exemplo acima, como:

**Exemplo 2:**



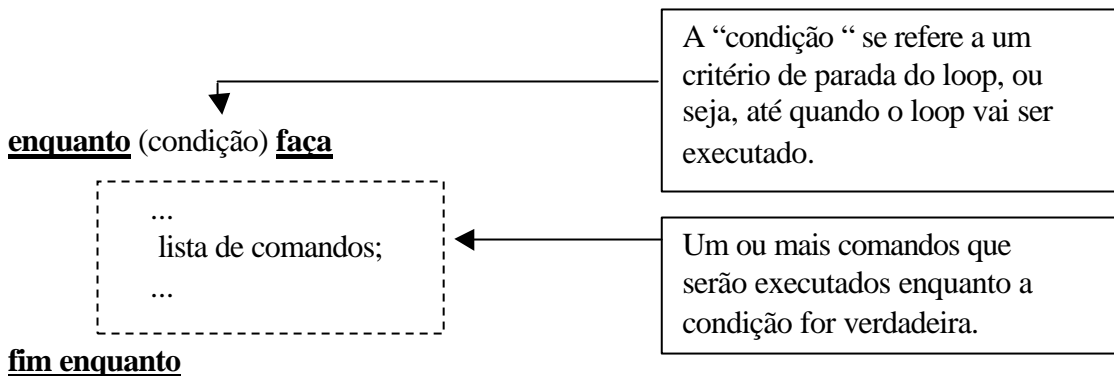
**Exemplo 3:**



Então, pode-se observar que as construções dos exemplos 2 e 3, representam as N repetições do exemplo 1. Em termos práticos da programação, a forma dos exemplos 2 e 3, de escrever ações que se repetem, são corretas. A forma de escrever as ações do exemplo 1 que se repetem é incorreta, apesar de levar o mesmo resultado, pois imagine reescrever as mesmas ações para 365 dias, ou mais...

**1.6.2.1. Comando: enquanto/ faça**

Em português, escreve-se o comando **enquanto** / **faça**, da forma apresentada abaixo. Note que se forma um bloco de comandos, delimitado entre o início e o fim do loop. Veja o exemplo:



Suponha os algoritmos abaixo que calculam o valor de  $x^{10}$ , sendo x fornecido pelo usuário. Em termos de programação, pode-se ver a diferença na escrita dos programas a seguir, com e sem o uso de um laço de repetição (loop):

Exemplo sem loop	Exemplo com loop
<pre> <u>inicio</u>      <u>inteiro</u> x,y;     <u>leia</u> (x);      y &lt;- x;     y &lt;- y * x;     y &lt;- y * x;     y &lt;- y * x;     y &lt;- y * x;     ...     ...     ...     y &lt;- y * x;     <u>escreva</u> (y);  <u>fim</u> </pre>	<pre> <u>inicio</u>      <u>inteiro</u> x,y,z;     <u>leia</u> (x);     y &lt;- x;     z &lt;- 1;      <u>enquanto</u> (z &lt; 10) <u>faça</u>         y &lt;- y * x;         z &lt;- z + 1;      <u>fim enquanto</u>     <u>escreva</u> (y);  <u>fim</u> </pre>

**Exemplos:**

a) O problema do loop infinito:

```

inicio

    inteiro I;
    I <- 0;

    enquanto (I < 5) faça
        escreva (I);

    fim enquanto

fim

```

Teste de Mesa	
inicio	I = 0
1ª iteração	I = 0
2ª iteração	I = 0
3ª iteração	I = 0
...	...
infinitas iterações	I = 0

Obs: O programa ficará travado, pois a condição de saída do loop nunca será satisfeita

b) Corrigindo o problema do loop infinito:

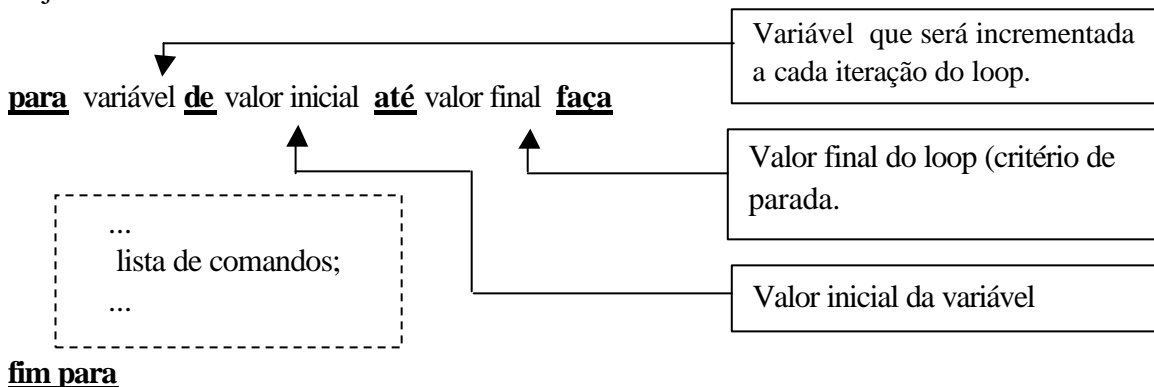
```

inicio
    inteiro I;
    I <- 0;
    enquanto (I < 5) faça
        I <- I + 1;
        escreva (I);
    fim enquanto
fim
    
```

Teste de Mesa	
inicio	I = 0
1ª iteração	I = 1
2ª iteração	I = 2
3ª iteração	I = 3
4ª iteração	I = 4
5ª iteração	I = 5
5 < 5 ?	sai do loop

**1.6.2.2. Comando: para / até / faça**

Em portugol, escreve-se o comando **para / até / faça**, da forma apresentada abaixo. Note que se forma um bloco de comandos, delimitado ente o início e o fim do loop. Veja a sintaxe do comando:



**Exemplos:**

a) Loop para/faça com passo crescente igual a 1.

```

inicio
    inteiro I;
    para I de 1 até 5 faça
        escreva (I);
    fim para
fim
    
```

Teste de Mesa	
inicio	
1ª iteração	I = 1
2ª iteração	I = 2
3ª iteração	I = 3
4ª iteração	I = 4
5ª iteração	I = 5
5 < 5 ?	sai do loop

Obs: No loop do tipo para/faça o valor da variável de controle do loop é incrementada automaticamente de 1 a cada loop.

b) Loop para/faça com passo diferente de 1 (incremento):

```
inicio
    inteiro I;
    para I de 1 até 5 passo 2 faça
        escreva (I);
    fim para
fim
```

Teste de Mesa	
inicio	
1ª iteração	I = 1
2ª iteração	I = 3
3ª iteração	I = 5
5 < 5 ?	sai do loop

Obs: No loop do tipo para/faça o valor da variável de controle do loop é incrementada automaticamente de 2 em 2 a cada loop.

c) Loop para/faça com passo decrescente:

```
inicio
    inteiro I;
    para I de 5 até 1 passo -2 faça
        escreva (I);
    fim para
fim
```

Teste de Mesa	
inicio	
1ª iteração	I = 5
2ª iteração	I = 3
3ª iteração	I = 1
1 < 1 ?	sai do loop

Obs: No loop do tipo para/faça o valor da variável de controle do loop é decrementada automaticamente de 2 em 2 a cada loop.

### Exercícios:

- 1 – Escreva um algoritmo para gerar uma PA de razão qualquer, com uma série de 10 termos.
- 2 – Modifique o exercício 5.1 para uma PA de N termos.
- 3 – Escreva um algoritmo para gerar a sequência de Fibonacci da forma abaixo, até o vigésimo termo: 1,1,2,3,5,8,13, ...
- 4 – Sejam dados P(X1,Y1) e Q(X2,Y2) dois pontos quaisquer no plano. Escreva um algoritmo que leia os pares de coordenada x e y e calcule a distância entre estes dois pontos.
- 5 – Escreva um algoritmo que gere uma tabela com a conversão de graus para Fahrenheit para Celsius e vice versa, com valores variando de 1 em 1 grau, de 0 a 100 graus Celsius.

## 2. PROGRAMAÇÃO EM LINGUAGEM C

### 2.1. INTRODUÇÃO A PROGRAMAÇÃO EM LINGUAGEM C

#### 2.1.1. Declaração de Variáveis

As diferenças entre os tipos de variáveis do português para o C são:

inteiro = **int**

real = **float, double**

caracter = **char**

lógico = **bool** (normalmente não é necessário tipos booleanos para testes lógicos)

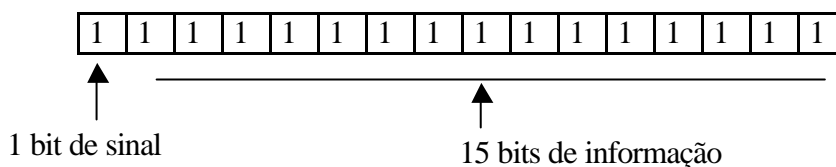
Para alterar a precisão dos valores podem ser utilizados modificadores:

C ANSI	Modificadores	Exemplos:
char	signed	<b>int</b> x ; (x é um inteiro com sinal - <b>signed</b> )
int	unsigned	<b>unsigned int</b> y; (x é um inteiro sem sinal - <b>unsigned</b> )
float	long	<b>long</b> z; (z é um inteiro com o dobro do tamanho de um <b>int</b> )
double	short	<b>short int</b> v; (v tem o mesmo tamanho de um <b>int</b> )

#### Exemplos:

a) **signed int** (ou simplesmente **int**):

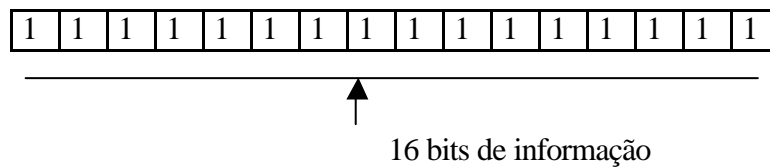
Tipo de variável que se refere a um número inteiro com sinal. Ou seja, se um variável **int** ocupa dois bytes na memória, então, o maior valor decimal que pode ser armazenado neste tipo de variável deve estar entre  $-32.767$  a  $32.767$ .



O primeiro bit (mais significativo) representa o sinal (0 positivo e 1 negativo). Então o maior valor decimal que pode ser armazenado em 15 bits é  $32.767$ .

b) **unsigned int;**

Tipo de variável que se refere a um número inteiro sem sinal. Ou seja, se um variável **int** ocupa dois bytes na memória, então, o maior valor decimal que pode ser armazenado neste tipo de variável deve estar entre 0 a 65.535.



O primeiro bit (mais significativo) não representa mais o sinal. Então o maior valor decimal que pode ser armazenado em 16 bits é 65.535.

O tamanho em bytes de cada tipo de variável no C, é apresentado na tabela abaixo.

Tipo	Tamanho em Bits	Faixa de valores
char	8	-127 a 127
unsigned char	8	0 a 255
signed char	8	-127 a 127
int	16	-32767 a 32767
unsigned int	16	0 a 65.535
signed int	16	mesmo que int
short int	16	mesmo que int
unsigned short int	16	mesmo que unsigned int
signed short int	16	mesmo que short int
long int	32	-2.147.483.647 a 2.147.483.647
signed long int	32	Mesmo que long int
unsigned long int	32	0 a 4.294.967.295
float	32	seis dígitos de precisão
double	64	dez dígitos de precisão
long double	80	dez dígitos de precisão

Fonte: C Completo e Total; Herbert Schildt.

**2.1.2. Comando de atribuição:**

O comando de atribuição em linguagem é dado pelo símbolo = (igual).

**Exemplo:**

```
float A, B, C;
A = 10;
B = 20;
C = A + B;
```

**2.1.3. Blocos de Comandos:**

Os blocos de comando, definidos no portugol pelas palavras **início/fim**, na linguagem C serão representados pelas { } (chaves).

**Exemplo:**

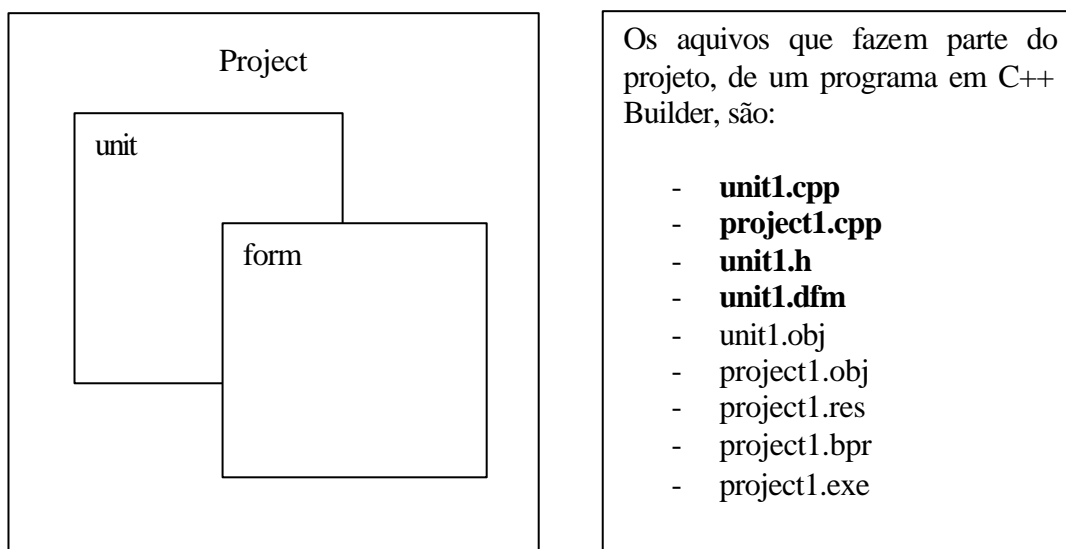
Portugol	Linguagem C
<pre><u>início</u>   real A, B, C;   A &lt;- 10;   B &lt;- 20;   c &lt;- A + B; <u>fim</u></pre>	<pre>{   float A, B, C;   A = 10;   B = 20;   C = A + B; }</pre>



## 2.2. BORLAND C++ BUILDER

### 2.2.1. O ambiente de desenvolvimento

O C++ Builder tem um ambiente de desenvolvimento integrado, com as ferramentas necessárias para a criação dos mais variados programas. Para o desenvolvimento de aplicações serão usados basicamente, formulários, alguns componentes e suas propriedades, "units", e bibliotecas.



Um programa escrito em ANSI C é representado por um arquivo que contém o código fonte e possui extensão “**C**” (*nomearquivo.c*). No C++ Builder o código fonte é escrito dentro de uma **unit** e o arquivo gravado possui a extensão “**.cpp**”. O projeto também tem extensão “**.cpp**”, e pode conter uma ou mais units, e um ou mais formulários (forms). Ao ser compilado é gerado um arquivo executável com a extensão “**.exe**”. O arquivo com extensão “**.h**”, armazena as definições dos recursos usados pela unit, e o arquivo com extensão “**.dfm**”, contém os recursos usados pelo programa.

Os arquivos “**.cpp**” são criados pelo programador, e os demais arquivos são criados automaticamente pelo compilador. Os arquivos destacados em negrito fazem parte do programa fonte, e serão necessários toda a vez que for preciso modificar o programa. Os demais são gerados a cada compilação.

### 2.2.2. A interface de desenvolvimento

Quando se inicia a criação de um programa (opção *new applicattion* do menu), são apresentados ao usuário, a janela do formulário (*form*), a barra de componentes (*component palette*) e a barra de propriedades e eventos (*object inspector*).

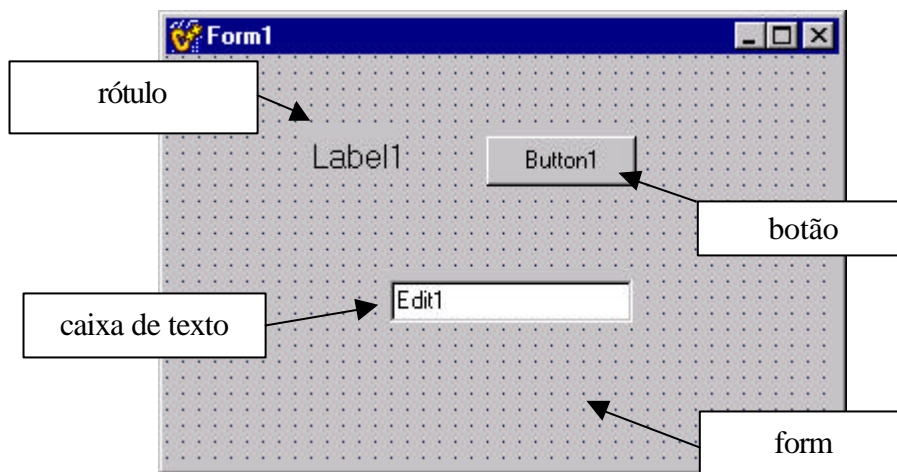
#### 3.2.2.1. Barra de Componentes

A barra de componentes apresenta todos os elementos que podem ser adicionados a um formulário, para a criação de um programa.



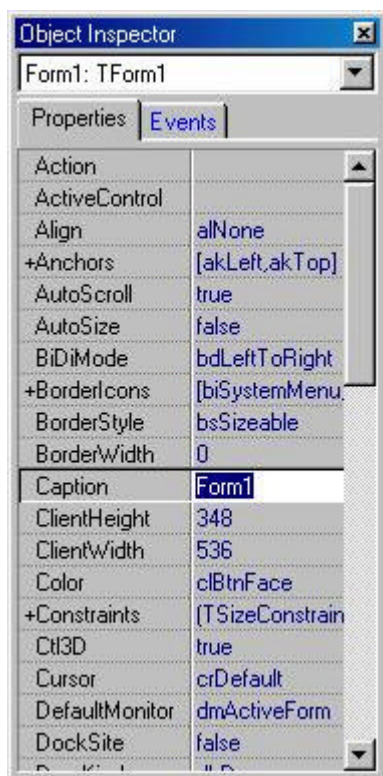
#### 2.2.2.2. Formulário (form)

O formulário (form) é a janela que irá receber os componentes (botões, edits, etc) que irão operar sobre o programa.



O formulário é a interface entre o programa e o usuário. É o meio pelo qual o usuário interage com o programa, seja inserindo dados através de uma caixa de texto, seja executando uma função ao clique de um botão, etc.

### 2.2.2.3. Barra de Propriedades



Esta janela apresenta as propriedades do componente com o "foco". As propriedades, ou eventos podem ser alterados nesta janela, quando da escrita do programa.

Por exemplo:

- a) para alterar o título do formulário, alterar a propriedade "**caption**";
- b) para alterar o nome do formulário alterar a propriedade "**name**";
- c) para alterar as cores do formulário alterar a propriedade "**color**".

As propriedades podem ser alteradas também durante a execução do programa, bastando referenciar o componente e a respectiva propriedade através do conector "->". Assim, no programa, pode-se escrever:

```
Form1->Caption = "programa1";
```

Cada componente possui uma lista de propriedades (no object inspector) que pode ser alterada de acordo com a necessidade do programador. Da mesma forma os eventos podem ser utilizados para executarem ações durante o uso do programa.

### 2.2.3. A criação de programas

#### **Enunciado:**

*Dado o algoritmo que calcula a soma de dois números, elabore um programa em C, que realize as operações representadas no algoritmo.*

Para resolver este problema, primeiramente é necessário conhecer as diferenças entre os comandos do português e da linguagem C, como apresentado a seguir:

Portugol	Linguagem C
<pre> <u>inicio</u>     <u>inteiro</u> a,b,c;     <u>leia</u>(a);     <u>leia</u>(b);     c = a + b;     <u>escreva</u>(c); <u>fim</u> </pre>	<pre> {     int a,b,c;     a = atoi(Edit1-&gt;Text.c_str());     b = atoi(Edit2-&gt;Text.c_str());     c = a + b;     Edit3-&gt;Text = c; } </pre>

Algumas diferenças ocorrem na leitura, escrita e atribuição de valores, como também em relação aos operadores relacionais, lógicos e aritméticos, como apresentado a seguir:

#### a) Entrada de Dados

`leia(n);` é escrito como `n = atoi(Edit1->Text.c_str());`

Onde,

**Edit1** é o nome do componente `EditText`;

**Edit1->Text** é a propriedade texto de **Edit1**;

**Edit1->Text.c\_str()** é o formalismo do Builder para leitura de string;

**atoi()** é a função do C para converter caracteres alfanuméricos (texto) em valor numérico do tipo inteiro.

Para usar a função `atoi()` é necessário incluir uma biblioteca do C:

```
#include <stdlib.h>
```

Obs:

- Note que Edit1 é o nome do componente e Text é uma das propriedades; esta especificamente diz respeito ao conteúdo da caixa de texto.
- Note também que Edit1 é o nome do EditText. Este nome será usado para todo o programa. Lembre que através da propriedade name, o nome pode ser alterado. Por exemplo, `Text01->Text = ...;`

**b) Atribuição**

`f <- n;` é escrito como `f = n;` (com o sinal de igualdade)

**c) Saída de Dados**

`escreva(f);` em C, é dado atribuindo um valor de resposta ao componente do formulário usado para apresentá-la, como por exemplo, um `EditBox`. Ou seja,

`Edit2->Text = f;` (a propriedade `Text` do `Edit2`, irá conter o valor de `f`).

**d) Operadores aritméticos em C**

Operação	Símbolo	Operação	Símbolo
Adição	+	Raiz quadrada	<code>sqrt()</code>
Subtração	-	Exponenciação	<code>pow()</code>
Multiplicação	*	Resto	%
Divisão	/		

**e) Operadores relacionais**

Operação	Símbolo	Operação	Símbolo
Maior que	>	Menor ou igual	<code>&lt;=</code>
Menor que	<	Igualdade	<code>==</code>
Maior ou igual	<code>&gt;=</code>	Diferença	<code>!=</code>

**2.2.4. Passos para Criar uma Aplicação em C**

A seguir será apresentada uma sequência de passos para a criação de programas em C, no ambiente do Borland C++ Builder.

### a) Abrindo o C++ Builder

No Windows, a partir do menu iniciar (ou atalho) selecione a opção "C++Builder 5". Ao ser iniciado, o ambiente apresenta os menus, barras de ferramentas, um formulário denominado **Form1** e uma área de texto, denominada **Unit1.cpp**. Estes elementos flutuam sobre a tela, e sua exibição pode ser alternada, apenas mudando o "foco".

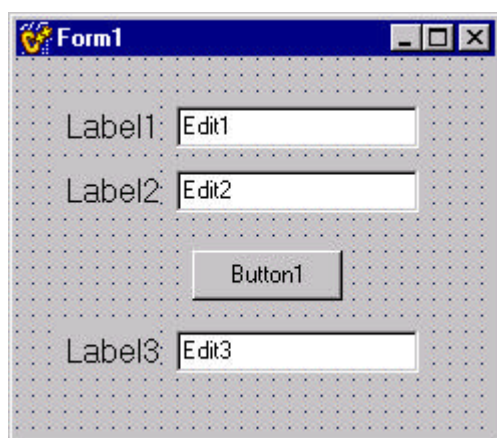
### b) Adicionando Formulário

Um formulário em branco (sem componentes) é automaticamente apresentado ao se criar uma aplicação. Se necessário, um formulário pode ser adicionado à aplicação através do botão "new form" como apresentado pelo menu abaixo.



### c) Inserindo Componentes no Formulário

O formulário representa a área de trabalho, onde será realizada a interface entre o programa e o usuário. Neste formulário devem ser inseridos os componentes da linguagem (botões, caixas de texto, caixas de verificação, etc). Para inserir um componente, é necessário escolher o componente desejado na barra de componentes e clicar na região do formulário onde quiser adicioná-lo.

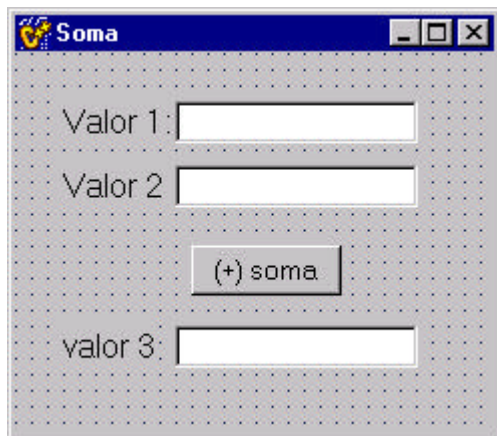


O formulário **Form1** contém:

- Edit Box (caixa de texto): **Edit1**
- Edit Box (caixa de texto): **Edit2**
- Edit Box (caixa de texto): **Edit3**
- label (rótulo): **Label1**
- label (rótulo): **Label2**
- label (rótulo): **Label3**
- Button (Botão): **Button1**

Ao serem incorporados ao Formulário, os componentes assumem nomes padronizados.

Os nomes podem ser alterados através das propriedades do *object inspector*. Então o formulário pode ficar como:



Para modificar o texto do label ou botão, deve-se alterar o conteúdo da propriedade **Caption**. Para Edits, alterar a propriedade **Text**. Lembre que a propriedade **Caption** não altera o nome do componente, apenas altera o texto que aparece sobre o componente. Assim, no programa os nomes continuam sendo `Edit1`, `Edit2`, ... etc. Para mudar o nome usar a propriedade **Name**.

#### d) Codificação do Programa

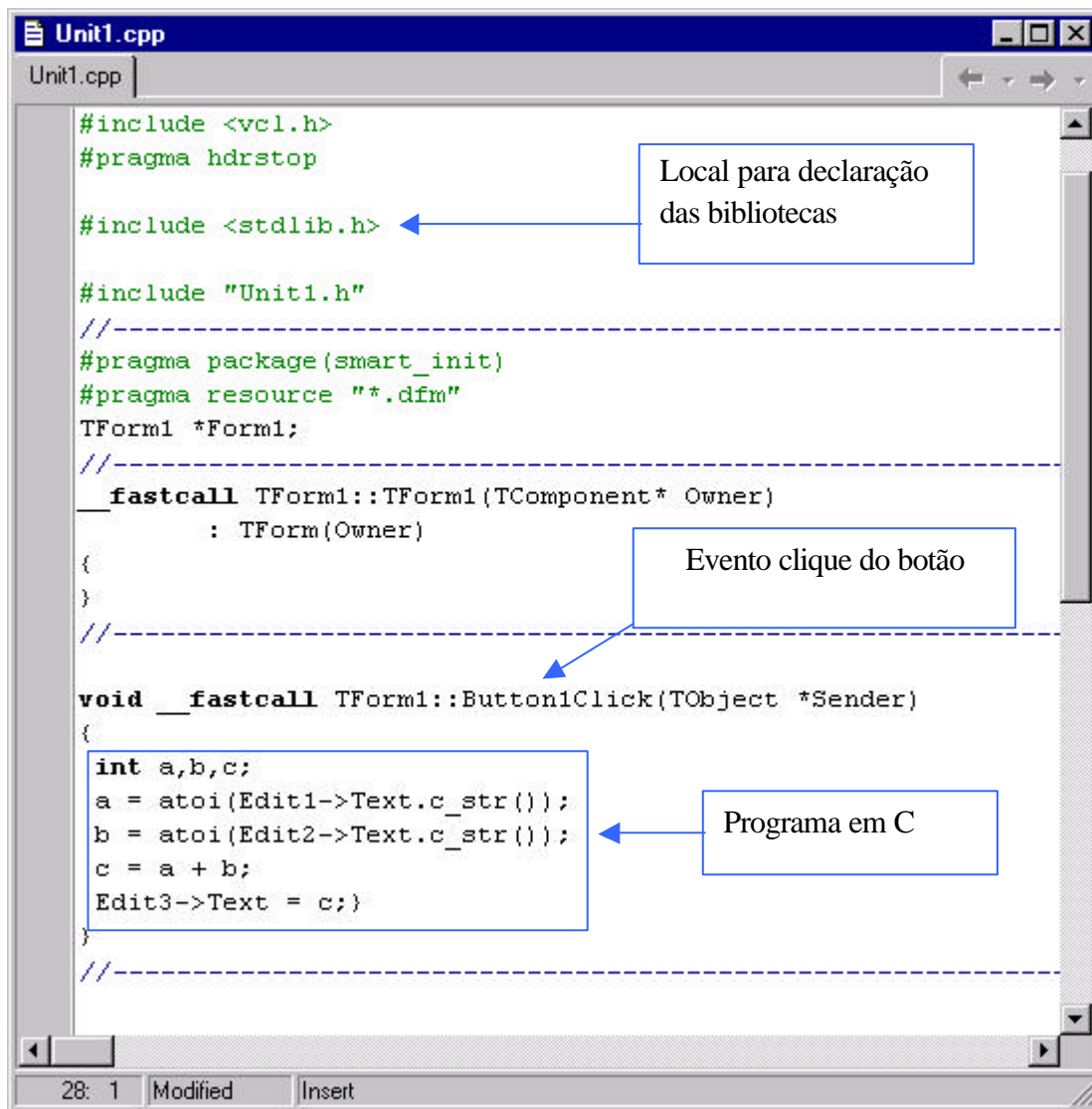
A janela de edição permite visualizar e alterar o código do programa de cada "Unit" como mostra a figura abaixo. Cada componente pode perceber a ocorrência de um evento, por exemplo, o clique em um botão. Então, pode-se associar ao componente **Button**, as linhas de código que executam as operações. Ou seja, para calcular a soma entre dois números, devemos fornecer um valor como parâmetro através de uma **EditBox**, e o cálculo do fatorial só será executado mediante o clique do botão.

Normalmente associa-se:

comandos, aos componentes do tipo **Button**;

entrada e saída de dados na tela, aos componentes **EditBox** ou **Label**;

parâmetros condicionais aos componentes **RadioButton** e **CheckBox**.



### **e) Compilando um Programa**

O C++ Builder deve compilar o programa para verificar a sintaxe, e gerar o código executável. Para compilar selecione a opção correspondente no menu, ou use as teclas de atalho Ctrl F9 ou F9.

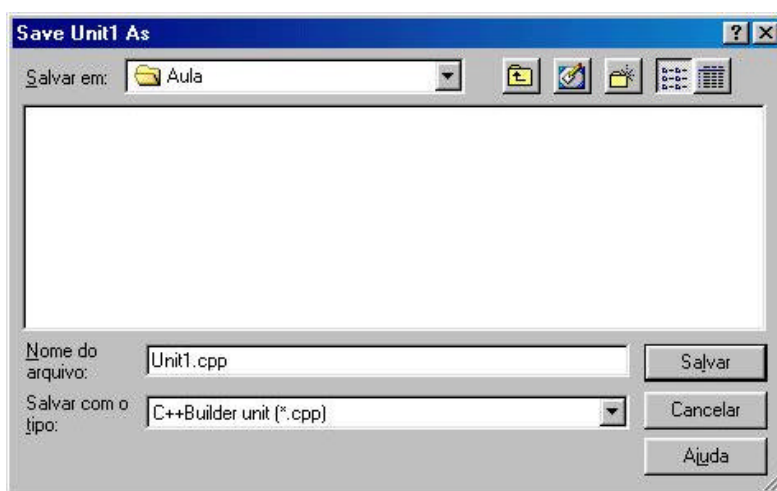
### **f) Executando um Programa**

Para executar o programa, utilize o comando RUN ou a tecla F9.



### g) Salvando o Programa

Para gravar o programa recomenda-se criar uma pasta, pois vários arquivos gerados pelo compilador devem ser armazenados. Se este local for conhecido, então facilitará a localização e manutenção dos programas. Use o comando "Save All" para que todos os arquivos sejam salvos. Uma janela, como abaixo, é apresentada. Forneça um nome para o arquivo **Unit1.cpp**. Em seguida deverá ser fornecido um nome para o arquivo de projeto **Project1.cpp**.



Obs:

- Como a unit e o projeto tem a mesma extensão (.cpp), não esqueça de dar nomes diferentes para os dois arquivos.
- Grave sempre no HD, pois o C++ gera vários arquivos durante a compilação.

### 2.2.5. Exercícios

1. Escreva um programa em C para somar e subtrair dois números fornecidos por um usuário, e apresentar o resultado usando um Editbox.
2. Modifique o exercício 3.1, para apresentar o resultado através de um Label;
3. Escreva um programa em C que calcule as raízes de uma equação de 2.º grau.
4. Escreva um algoritmo e um programa em C que conte quantas vezes um botão foi pressionado, e apresente o resultado em um EditBox.

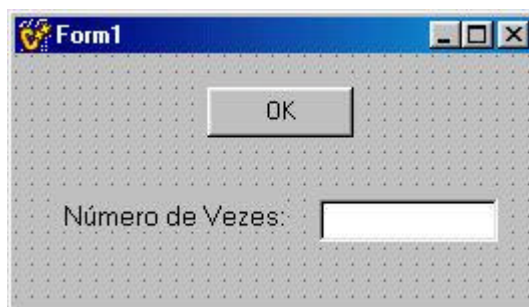
## 2.3. ESCOPO DE VARIÁVEIS

### 2.3.1. Variáveis locais

As variáveis devem existir (ser declaradas) antes de serem usadas no programa. Existem duas formas de declaração de variáveis. As variáveis locais são declaradas dentro de um procedimento ou função, ou associadas a um evento, como por exemplo, um botão no formulário. Estas variáveis são utilizadas dentro do bloco de comandos do botão. Estas variáveis não são acessíveis por outras partes do programa, por exemplo, associadas a um segundo botão.

### Exemplo

Escreva um algoritmo e um programa em C (usando C++ Builder), para contar o número de vezes que foi apertado o botão [OK] do formulário abaixo. Implementar este exemplo e analisar o resultado obtido.



*Dentro do botão **OK**:*

```
{  
    int vezes = 0; // variável local  
    vezes = vezes + 1; // incrementa o número de vezes  
    Edit1->Text = vezes; // apresenta resultado  
}
```

### 2.3.2. Variáveis globais

As variáveis globais devem ser declaradas em um local que possa ser conhecida por todos os componentes do programa, normalmente no início do programa (da unit). Uma variável global pode ser usada por qualquer componente do programa. Por exemplo, dois botões que realizam operações diferentes têm acesso às mesmas variáveis.

*No início da UNIT.CPP*

```
int vezes = 0; // variável global
```

*Dentro do botão OK:*

```
{  
    vezes = vezes + 1; // conta o num. de vezes  
    Edit1->Text = vezes; // apresenta resultado  
}
```

#### ***Quando usar variáveis locais ou globais?***

Normalmente usa-se variáveis globais quando um valor deve ser conhecido e manipulado em várias partes do programa. Normalmente usa-se variáveis locais quando a variável tem uma função específica dentro de um bloco de comandos (por exemplo, contador de loop).

Talvez a diferença mais importante entre os dois tipos, é que os conteúdos de variáveis locais estão alocadas na memória somente durante a execução do bloco de programa que as necessita. Assim, depois de usada a memória é liberada e quando necessário é alocada novamente.

Os conteúdos de variáveis globais estão disponíveis enquanto o programa estiver sendo executado. A região de memória alocada para cada variável, permanece bloqueada durante toda a execução do programa.

## 2.4. Desvio condicional em C

O comando **if - else** permite expressar decisões que alteram o fluxo de execução de um programa. A sintaxe é dada por:

### 2.4.1. Desvio Condicional Simples

Portugol	Bloco com N comandos	Apenas um comando
<pre> ... se (condição) então     comando; fim se ...                     </pre>	<pre> ... if (condição) {     comando 1;     comando 2;     ...     comando n; } ...                     </pre>	<pre> ... if (condição)     comando; ...                     </pre> <div style="border: 1px solid blue; padding: 5px; margin-top: 10px;"> <p>Com apenas um comando <u>não é</u> necessário abrir um bloco com chaves { }.</p> </div>

### 2.4.2. Desvio Condicional Composto

Portugol	Bloco com N comandos	Apenas um comando
<pre> se (condição) então     comando 1; senão     comando 2; fim se                     </pre>	<pre> if (condição) {     comando 1;     comando 2;     comando 3; } else {     comando n-1;     comando n; }                     </pre>	<pre> if (condição)     comando 1; else     comando 2;                     </pre> <div style="border: 1px solid blue; padding: 5px; margin-top: 10px;"> <p>Com apenas um comando <u>não é</u> necessário abrir um bloco com chaves { }.</p> </div>

### 2.4.3. If's Aninhados

<pre> <u>se</u> (condição) <u>então</u>     <u>se</u> (condição) <u>então</u>         comando 1;     <u>fim se</u>     <u>se</u> (condição) <u>então</u>         comando 2;     <u>senão</u>         comando 3;     <u>fim se</u> <u>senão</u>         comando 4; <u>fim se</u>         </pre>	<pre> <b>if</b> (condição) {     <b>if</b> (condição) comando 1;     <b>if</b> (condição) comando 2;     <b>else</b> comando 3; } <b>else</b>     comando 4;         </pre>	<pre> <b>if</b> (condição)     comando 1; <b>else</b>     <b>if</b> (condição)         comando 2;     <b>else</b>         <b>if</b> (condição)             comando 4;         </pre>
--	---	--

### 2.4.4. Exemplo

Escreva um algoritmo em português e o respectivo programa em C, para calcular as raízes de uma equação do 2.o grau:  $Ax^2 + Bx + C$ .

#### Solução sem o uso de “if”.

- Algoritmo em Português:

```

Início
    inteiro: A, B, C;
    real: x1, x2;
    leia(A, B, C);
    x1 <- (-B + raiz(B^2 - 4*A*C))/ 2*A;
    x2 <- (-B - raiz(B^2 - 4*A*C))/ 2*A;
    escreva(x1, x2);
fim
        
```

**- Programa em Linguagem C++ Builder**

```

#include <stdlib.h> // contém atoi()
#include <math.h>   // contém pow() e sqrt()
// no botão OK:
{
    int A, B, C;
    float x1, x2;
    A = atoi(Edit1->Text.c_str());
    B = atoi(Edit2->Text.c_str());
    C = atoi(Edit3->Text.c_str());
    x1 = (-B + sqrt(pow(B,2) - 4*A*C))/ 2*A;
    x2 = (-B - sqrt(pow(B,2) - 4*A*C))/ 2*A;
    Edit4->Text = x1;
    Edit5->Text = x2;
}

```

**b) Solução com o uso de “if”.****- Algoritmo em Portugol:**iniciointeiro: A, B, C, D;real: x1, x2;leia(A, B, C);D <- (B<sup>2</sup> - 4\*A\*C);se (D >= 0) então

x1 &lt;- (-B + raiz(D)/ 2\*A;

x2 &lt;- (-B - raiz(D)/ 2\*A;

escreva(x1, x2);senãoescreva("Delta < 0");fim sefim

**- Programa em Linguagem C++ Builder**

```

#include <stdlib.h> // contém atoi()
#include <math.h>   // contém pow()
// no botão OK:
{
    int A, B, C, D;
    float x1, x2;
    A = atoi(Edit1->Text.c_str());
    B = atoi(Edit2->Text.c_str());
    C = atoi(Edit3->Text.c_str());
    D = pow(B,2) - 4*A*C;
    if (D >= 0)
    {
        x1 = (-B + sqrt(D))/ 2*A;
        x2 = (-B - sqrt(D))/ 2*A;
        Edit4->Text = x1;
        Edit5->Text = x2;
    }
    else
        ShowMessage("Delta < 0");
}

```

**2.4.5. Exercício**

1. Escreva um programa para converter graus Celsius para Fahrenheit, e vice versa, a partir de uma temperatura fornecida por um usuário, usando o form A) e o form B).

form A

form B

## 2.5. Laços de repetição em C

### 2.5.1. Loop Para/Faça (for)

**Exemplo :**

Desenvolva um programa que gere uma tabela de conversão de temperatura de graus Fahrenheit para graus Celcius.

Portugol	Linguagem C++ Builder
<pre> <u>inicio</u>     inteiro x;     real C, F;     para F de 0 até 100 faça         C &lt;- (5 * (F-32)) / 9;     fim para <u>fim</u>                     </pre>	<pre> {     int x;     float C, F;     for (F = 0; F &lt; 100; F++ )     {         C = (5 * (F-32)) / 9;     } }                     </pre>

### 2.5.2. Loop Enquanto/Faça (while)

Portugol	Linguagem C++ Builder
<pre> <u>inicio</u>     inteiro x;     real C, F;     F &lt;- 0;     enquanto F &lt; 100 faça         C &lt;- (5 * (F-32)) / 9;         F &lt;- F + 1;     fim para <u>fim</u>                     </pre>	<pre> {     int x;     float C, F;     F = 0;     while (F &lt; 100)     {         C = (5 * (F-32)) / 9;         F++; // F = F + 1;     } }                     </pre>



**2.5.3. Loop Faça/Enquanto (do/while)**

Portugol	Linguagem C++ Builder
<pre> <u>inicio</u>     <u>inteiro</u> x;     <u>real</u> C, F;     F &lt;- 0;     <u>faça</u>         C &lt;- (5 * (F-32)) / 9;         F &lt;- F + 1;     <u>enquanto</u> F &lt; 100; <u>fim</u>                 </pre>	<pre> {     <b>int</b> x;     <b>float</b> C, F;     F = 0;     <b>do</b>     {         C = (5 * (F-32)) / 9;         F++; // F = F + 1;     } <b>while</b> (F &lt; 100) }                 </pre>

**2.5.4. Exemplo**

Escreva um programa em C para calcular o fatorial de um número inteiro e positivo fornecido pelo usuário do programa.

```

void __fastcall TForm1::Button1Click(TObject *sender)
{
    int n, f, x;
    n = atoi(Edit1->Text.c_str());
    f = n;
    for ( x = n ; x > 1 ; x-- )
    {
        f = f * (x-1);
    }
    Edit2->Text = f;
}
                
```



### 2.5.5 Exercícios

1. Altere o loop **for** do exemplo anterior, para os loops **while** e **do/while**.
2. Escreva o algoritmo e o respectivo programa em C para um programa que conte a quantidade de números pares e ímpares digitados por um usuário. O usuário pode digitar quantos números quiser, e pode encerrar o programa quando desejar.
3. Escreva o algoritmo e o respectivo programa em C, para um programa que conte a quantidade de números primos digitados por um usuário. O usuário pode digitar quantos números quiser, e pode encerrar o programa quando desejar.

## 2.6. PROCEDIMENTOS EM C

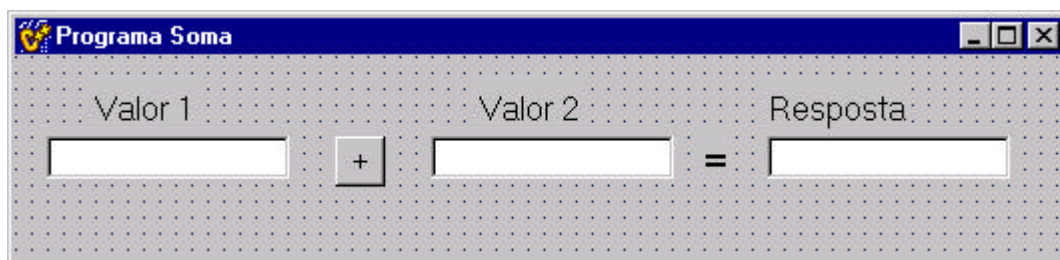
### 2.6.1. Definição

A utilização de procedimentos permite que um conjunto de comandos possa ser usado repetidas vezes dentro de um programa, sem a necessidade de reescrever o código várias vezes. Um bloco de comandos é associado a um nome (nome do procedimento); sempre que for necessário executar estes comandos, basta chamar o nome do procedimento.

### 2.6.2. Exemplo 1

Desenvolva um programa que calcule a soma de dois valores reais fornecidos pelo usuário.

- **Formulário:**

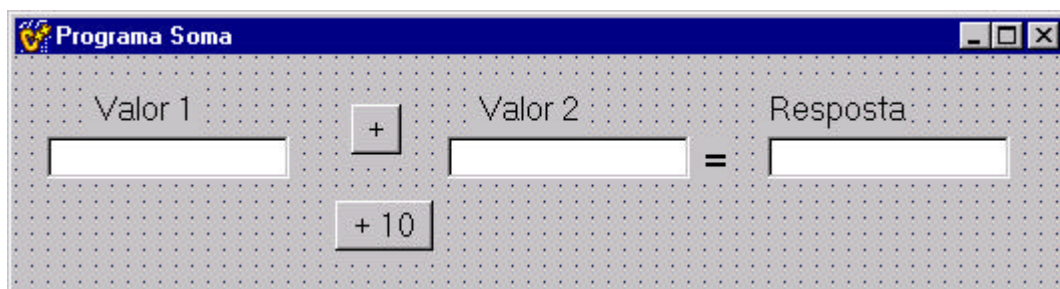


- **Dentro do Botão [+]:**

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    float valor1, valor2, resp;
    valor1 = atof(Edit1->Text.c_str());
    valor2 = atof(Edit2->Text.c_str());
    resp = valor1 + valor2;
    Edit3->Text = resp;
}
```

Agora suponha que o usuário queira somar 10 a cada valor1 fornecido. Pode-se criar um novo botão para realizar esta operação como apresentado abaixo:

**- Formulário:**



**- Dentro do Botão [+ 10]:**

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    float valor1, resp;

    valor1 = atof(Edit1->Text.c_str());
    resp = valor1 + 10;
    Edit3->Text = resp;
}
```

Observando-se os dois botões, percebe-se facilmente que os dois realizam uma operação de soma de dois valores ( $\text{valor1} + \text{valor2}$  ou  $\text{valor1} + 10$ ).

Cada uma das operações de soma terá um código correspondente em linguagem de máquina. Ou seja, dois trechos em linguagem de máquina que irão realizar a mesma operação: somar dois valores. Pode-se otimizar este programa escrevendo uma única vez a operação de soma de forma que possa ser acessada pelos dois botões.

Então podemos criar um procedimento que realiza a soma, da seguinte forma:

```

//-----
#include <vcl.h>
#pragma hdrstop
#include <stdlib.h>
#include "Unit1.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//-----

float resp;
//-----

void Soma(float a, float b)
{
    resp = a + b;
}
//-----

__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//-----

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    float valor1, valor2;
    valor1 = atof(Edit1->Text.c_str());
    valor2 = atof(Edit2->Text.c_str());
    Soma(valor1, valor2);
    Edit3->Text = resp;
}
//-----

void __fastcall TForm1::Button2Click(TObject *Sender)
{
    float valor1;
    valor1 = atof(Edit1->Text.c_str());
    Soma(valor1, 10);
    Edit3->Text = resp;
}
//-----

```

**1**-Bibliotecas e diretivas de compilação

**2**-Espaço para declaração de variáveis globais

**3**-Espaço para declarar procedimentos e funções ou seus protótipos.

**4**-Trecho de programa do botão [+] e a chamada do procedimento Soma.

**5**-Trecho de programa do botão [+ 10] e a chamada do procedimento Soma.

Da listagem anterior pode-se observar que a observação número:

- 1 – identifica o local aonde são adicionadas as bibliotecas e diretivas do programa. Sempre no início da Unit,
- 2 –identifica o local para declaração de variáveis globais. As variáveis globais devem existir antes de serem usadas pelo programa, pois isto vem antes; note que a variável resp é usado dentro do procedimento, e também dentro de cada botão.
- 3 –identifica o local para escrever os procedimentos (ou protótipos). Da mesma forma que a variável global, o procedimento precisa primeiro existir para que possa ser usado pelo resto do programa. Pode ser escrito o procedimento inteiro ou apenas o protótipo; neste caso o bloco do procedimento pode ser escrito em qualquer parte do programa. (menos dentro de eventos do formulário).
- 4- identifica a chamada do procedimento. No momento em que precisa-se calcular a soma, chama-se o procedimento. Note que existe passagem de parâmetros. O conteúdo de valor1 é passado para a variável a do procedimneto e o conteúdo de valor2 é passado para a variável b do procedimento. Observe também que valor1 e valor2 são variáveis locais do botão, portanto não podem ser acessadas pelo procedimento; por isto é necessário realizar passagem de parâmetros. As variáveis a e b são locais do procedimento (só existem dentro do procedimento), mas a resposta é atribuída a resp, que é uma variável global, portanto pode ser acessada por todo o programa.
- 5- igual ao item 4, com o valor constante 10 em vez da variável valor2.

### **2.6.3. Protótipo**

Dado o exemplo anterior, formalmente pode-se criar o protótipo de um procedimento com apresentado a seguir:

NomeDoProcedimento(lista de parâmetros);

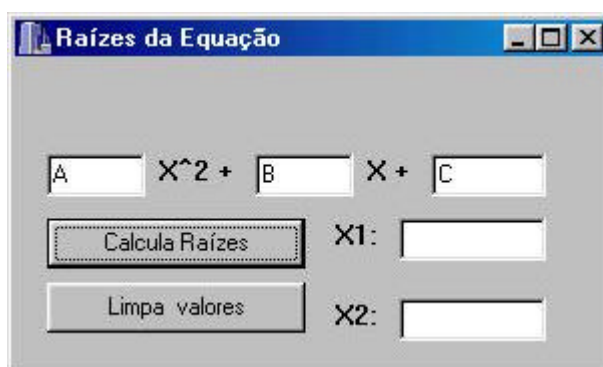
Sempre declarando o procedimento como global, ele pode ser acessado em qualquer parte do programa.

### 2.6.4. Parâmetros

Uma ou mais variáveis de tipos iguais ou diferentes que são utilizadas pelo procedimento para a realização de alguma operação específica. Durante a execução sequencial de um programa, um procedimento pode ser chamado e pode receber alguns valores para serem operados.

### 2.6.5. Exemplo 2

Dado o programa em C, que calcula as raízes de uma equação do segundo grau, modifique a estrutura para a utilização de um procedimento para limpar a tela (conteúdo das variáveis e caixas de texto).



#### a) Sem Procedimento

```
// declaração de variáveis globais
int A, B, C, D; // parâmetros da equação
float x1, x2; // raízes da equação

. . .

//-----
// botão para o cálculo das raízes
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    A = atoi(Edit1->Text.c_str()); // entrada dos valores
    B = atoi(Edit2->Text.c_str());
    C = atoi(Edit3->Text.c_str());
}
```

```

D = pow(B,2) - 4*A*C; // calcula delta
if ( D >= 0 )          // testa se existe raízes
{
    x1 = (-B + sqrt(D))/ 2*A;
    x2 = (-B - sqrt(D))/ 2*A;
    Edit4->Text = x1; // apresenta resposta
    Edit5->Text = x2;
}
else
{
    ShowMessage("Delta < 0");
    Edit1->Text = ""; // limpa Caixas de Texto
    Edit2->Text = "";
    Edit3->Text = "";
    Edit4->Text = "";
    Edit5->Text = "";
    A=0;B=0;C=0;D=0; // limpa valores da equação
    x1=0;x2=0;      // limpa resultado
}
}
//-----
// botão para limpar os valores da tela
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    Edit1->Text = ""; // limpa Caixas de Texto
    Edit2->Text = "";
    Edit3->Text = "";
    Edit4->Text = "";
    Edit5->Text = "";
    A=0;B=0;C=0;D=0; // limpa valores da equação
    x1=0;x2=0;      // limpa resultado
}
//-----

```

No exemplo acima , note que os trechos de programa escritos em vermelho se repetem. Então podemos transformar o trecho repetido, em um procedimento.



**b) Com Procedimento**

```

// declaração de variáveis globais
int A, B, C, D;
float x1, x2;
//-----
// procedimento para limpar a tela
void LimpaTela(void)
{
    Form1->Edit1->Text = ""; // limpa Caixas de Texto
    Form1->Edit2->Text = "";
    Form1->Edit3->Text = "";
    Form1->Edit4->Text = "";
    Form1->Edit5->Text = "";
    A=0;B=0;C=0;D=0; // limpa valores da equação
    x1=0;x2=0;      // limpa resultado
}

//-----
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    A = atoi(Edit1->Text.c_str());
    B = atoi(Edit2->Text.c_str());
    C = atoi(Edit3->Text.c_str());
    D = potencia(B,2) - 4*A*C;
    if ( D >= 0 )
    {
        x1 = (-B + sqrt(D))/ 2*A;
        x2 = (-B - sqrt(D))/ 2*A;
        Edit4->Text = x1;
        Edit5->Text = x2;
    }
    else
    {
        ShowMessage("Delta < 0");
        LimpaTela();
    }
}
}

```

```

//-----
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    LimpaTela();
}
//-----

```

Neste exemplo, foi criado um procedimento com a função de limpar os edits. Note que neste caso não existe passagem de parâmetros (void) porque o procedimento não precisa receber nenhum valor.

## 2.7. FUNÇÃO EM C

### 2.7.1. Definição

Uma função é um procedimento que retorna um valor (um resultado). Normalmente é um bloco de comandos que executa uma tarefa específica, e após executá-la devolve um resultado das operações realizadas. Este resultado pode ser atribuído a uma variável em qualquer parte do programa principal. Tanto uma função, como um procedimento podem ser chamados (executados) várias vezes, durante o funcionamento do programa.

### 2.7.2. Declaração

```

tipo NomeFuncao(tipo do parametro 1, tipo do parametro2, ... , tipo do parametroN)
{
    ...
    return valor;
}

```

### 2.7.3 Parâmetros e retorno

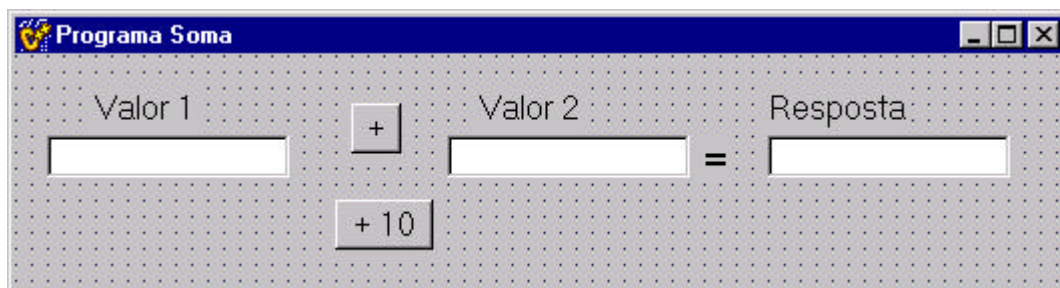
Parâmetros são os valores passados para a função que vão sofrer algum tipo de modificação durante a execução dos comandos da função. Sempre deve ser retornado um valor (**return**). A

variável que vai retornar o valor deve ser do mesmo tipo da função. Observe que a função sempre tem um tipo, ou seja, uma função pode ser do tipo inteiro, real, etc. Quando a função não tem um tipo (**void**), então ela não pode retornar um valor, ou seja passa a ser um procedimento.

#### 2.7.4. Exemplo 1

Desenvolva um programa que calcule a soma de dois valores reais fornecidos pelo usuário.

- **Formulário:**



No exemplo de procedimento os conteúdos das variáveis locais eram trocados via passagem de parâmetros, mas a variável de resposta resp, passou a ser global a fim de receber a soma dos conteúdos passados para a e b.

Agora, se a variável resp não for mais declarada como global, ou seja, passar a ser local de cada botão, então pode-se reescrever o procedimento na forma de uma função, como apresentado a seguir:

```

//-----
#include <vcl.h>
#pragma hdrstop
#include <stdlib.h>
#include "Unit1.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//-----
//-----
void Soma(float a, float b)
{
    float r;
    r = a + b;
    return r;
}
//-----
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//-----
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    float valor1, valor2, resp;;
    valor1 = atof(Edit1->Text.c_str());
    valor2 = atof(Edit2->Text.c_str());
    resp = Soma(valor1, valor2);
    Edit3->Text = resp;
}
//-----
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    float valor1, resp;
    valor1 = atof(Edit1->Text.c_str());
    resp = Soma(valor1, 10);
    Edit3->Text = resp;
}

```

**1**-Bibliotecas e diretivas de compilação.

**2**-Espaço para declaração de variáveis globais.

**3**-Espaço para declarar procedimentos e funções ou seus protótipos.

**4**-Trecho de programa do botão [+] e a chamada da função Soma.

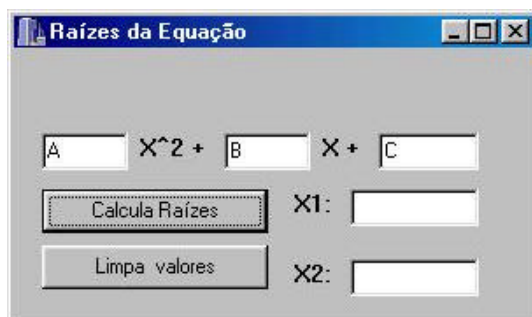
**5**-Trecho de programa do botão [+ 10] e a chamada da função Soma.

Da listagem anterior pode-se observar que a observação número:

- 1 – identifica o local aonde são adicionadas as bibliotecas e diretivas do programa. Sempre no início da Unit,
- 2 – identifica o local para declaração de variáveis globais. Note que neste exemplo, não utiliza-se mais variáveis globais;
- 3 – identifica o local para escrever os procedimentos (ou protótipos). Da mesma forma que a variável global, o procedimento precisa primeiro existir para que possa ser usado pelo resto do programa. Pode ser escrita toda a função ou apenas o protótipo; neste caso o bloco do procedimento pode ser escrito em qualquer parte do programa. (menos dentro de eventos do formulário).
- 4- identifica a chamada da função. No momento em que precisa-se calcular a soma, chama-se o procedimento. Note que existe passagem de parâmetros. O conteúdo de valor1 é passado para a variável a do procedimneto e o conteúdo de valor2 é passado para a variável b do procedimento. Observe também que valor1 e valor2 são variáveis locais do botão, portanto não podem ser acessadas pelo procedimento; por isto é necessário realizar passagem de parâmetros. As variáveis a e b são locais do procedimento como também agora, a variável r. para que o valor de r possa ser conhecido pelo restante do programa a função deve retornar o conteúdo de r, através do comando return. A variável local resp recebe o valor retornado por r.
- 5- igual ao item 4, com o valor constante 10 em vez da variável valor2.

### 2.7.5. Exemplo 2

Modifique o programa de cálculo das raízes, para o uso de uma função que calcule o valor da exponencial de um número qualquer ( $x^y$ ), sem o uso do comando pow().



```

#include <vcl.h>
#pragma hdrstop
#include "raizes.h"
#include <stdlib.h> // contém atoi()
#include <math.h>   // contém pow()
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

//-----
// declaração de variáveis globais
int A, B, C, D;
float x1, x2;
//-----

__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{ }
//-----

void LimpaTela(void)
{
    Form1->Edit1->Text = ""; // limpa Caixas de Texto
    Form1->Edit2->Text = "";
    Form1->Edit3->Text = "";
    Form1->Edit4->Text = "";
    Form1->Edit5->Text = "";
    A=0;B=0;C=0;D=0; // limpa valores da equação
    x1=0;x2=0;      // limpa resultado
}
//-----

// função que realiza o cálculo da potência
long potencia(int base, int exp)
{
    int x; // conta o número de vezes a ser multiplicado
    long resp = base;
    for (x=1; x<exp; x++)
    {
        resp = resp * base;
    }
    return resp;
}

```

```

//-----
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    A = atoi(Edit1->Text.c_str());
    B = atoi(Edit2->Text.c_str());
    C = atoi(Edit3->Text.c_str());
    D = potencia(B,2) - 4*A*C; // chamada da função
    if ( D >= 0 )
    {
        x1 = (-B + sqrt(D))/ 2*A;
        x2 = (-B - sqrt(D))/ 2*A;
        Edit4->Text = x1;
        Edit5->Text = x2;
    }
    else
    {
        ShowMessage("Delta < 0");
        LimpaTela(); // chamada do procedimento
    }
}
//-----

void __fastcall TForm1::Button2Click(TObject *Sender)
{
    LimpaTela(); // chamada do procedimento
}
//-----

```

### **2.7.6. Exercícios**

1. Escreva e teste o funcionamento dos exercícios acima.
2. Modifique o programa do cálculo do fatorial para uma função que calcule o valor do fatorial de um número.
3. Desenvolva um programa que permita realizar operações de adição e subtração de dois valores fornecidos pelo usuário, através de uma função.

## 2.8. Incrementos e Decrementos

Incrementar e decrementar variáveis são tarefas muito comuns em programação. A linguagem C oferece alguns comandos para realizar estas operações.

<p><b>Exemplo A:</b></p> <pre>{ <b>int</b> total, valor; valor = 0; total = valor + 1; //conteúdo de <b>valor</b> mais 1 ... }</pre>	<p><b>Exemplo B:</b></p> <pre>{ <b>int</b> total, valor; valor = 0; total = valor++; //conteúdo de <b>valor</b> mais 1 ? ... }</pre>
--	--

### 2.8.1. Incremento/Decremento a Posteriori

Neste caso, primeiro é feita a operação de atribuição e depois incrementa/decrementa a variável.

**Exemplo:**

Incremento	Decremento
<pre>{ <b>int</b> total; <b>int</b> valor; valor = 1; total = valor++; /** neste exemplo total recebe valor e depois da atribuição é somado 1 a variável valor **/ ... }</pre>	<pre>{ <b>int</b> total; <b>int</b> valor; valor = 1; total = valor--; /** neste exemplo total recebe valor e depois da atribuição é subtraído 1 da variável valor **/ ... }</pre>



**2.8.2. Incremento/Decremento a Priori**

Neste caso, primeiro é incrementado o valor, e depois é atribuído a variável.

**Exemplo:**

Incremento	Decremento
<pre>{ <b>int</b> total; <b>int</b> valor; valor = 1; total = ++valor; /** neste exemplo valor é incrementado e depois é feita a atribuição **/ ... }</pre>	<pre>{ <b>int</b> total; <b>int</b> valor; valor = 1; total = --valor; /** neste exemplo valor é decrementado e depois é feita a atribuição **/ ... }</pre>

**2.8.3. Exercício**

Qual o conteúdo das variáveis total e valor após a execução dos seguintes comandos?

```
{
int valor, total;
valor = 3;
total = 5;
total = valor++;
total = ++valor;
total = valor--;
total = --valor;
}
```

## 2.9. Atribuição Composta

Através da combinação de operadores pode-se fazer atribuições das seguintes formas:

### **Exemplo:**

```

total += valor; // total = total + valor;
valor += 1;    // valor = valor + 1;
valor *= 10;   // valor = valor * 10;

```

### 2.9.1. Exercício

Qual o valor das variáveis `valor` e `total` após a execução dos seguintes comandos?

```

{
  int valor = 3;
  int total = 5;
  total *= ++valor;
}

```

## 2.10. Atribuição Múltipla

Depois de declaradas as variáveis podem-se ter comandos de atribuição da seguinte forma:

```
total = valor = resposta = 0; // atribui 0 a todas as variáveis
```

### 2.10.1. Exemplo

```

{
  int resposta, valor, total;
  resposta=total=0;
  valor=4;
  reposta = total + valor;
}

```

ou,

```

{
  int resposta, valor, total;
  resposta=total=0;
  reposta = total + (valor = 4);
}

```

## 2.11. Operador Interrogação (?)

Este operador substitui sentenças do tipo **if/else**.

O trecho de programa:	pode ser escrito como:
<pre>... if ( x &gt;= 0 )     y = -1; else     y = 1; ...</pre>	<pre>... y = (x &gt;= 0) ? -1 : 1; // se, por exemplo x=10, y = -1; ...</pre>

## 2.12. Números Aleatórios

O comando **random()**, gera um número aleatório (randômico), inteiro e positivo obtido diretamente do computador.

### 2.12.1. Sintaxe do Comando

```
int nome_variavel = random(100);
```

Neste caso, uma variável do tipo **int** recebe um valor aleatório gerado pela função **random(100)**. O valor 100 significa que o valor gerado estará compreendido entre 0 e 99. Para usar o comando **random**, deve-se adicionar a biblioteca [stdlib.h](#).

### 2.12.2. Exemplo

```
{
int a;
a = random(10); // gera um valor aleatório entre 0 e 9
Edit1->Text = a;
}
```

## 2.13 comando switch/case

O comando switch/case pode ser usado para substituir uma sequência de **if/else** aninhados. Os comandos case realizam a escolha entre possíveis valores para uma variável, fornecida como parâmetro no comando switch.

### 2.13.1. Sintaxe do comando

**switch** (*variável*)

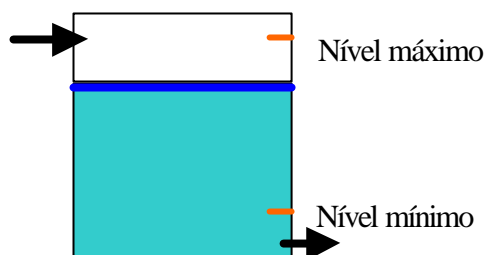
```
{
  case valor_contante:{
      comando 1;
      comando 2;
      break;
  }
  case valor_constante:{
      comando 3;
      comando 4;
      break;
  }
  default: comando 5;
}
```

O valor proveniente da variável é testado em cada **case**, sequencialmente do primeiro ao último. Caso, o *valor\_constante* seja igual ao da variável, então a execução do programa entra no **case**, e executa os comandos até encontrar um **break**.

Se nenhum case for satisfeito, automaticamente a execução do programa segue pela opção **default**.

### 2.13.2. Exemplo

Desenvolva um programa que apresente informação sobre o nível de um reservatório quando atingir o nível máximo e o nível mínimo. Os valores de nível devem ser aleatórios.



**Dentro do botão:**

```
{
int x;
x = random(10);
switch(x)
{
  case 0 :{
      Edit1->Text = "Reservatório vazio";
      break;
  }
  case 9 :{
      Edit1->Text = "Reservatório cheio";
      break;
  }
  default: ShowMessage("Reservatório Ok");
}
}
```

## 2.14. Timer

A geração de números aleatórios é muito utilizada em ambiente de simulação. No caso do exemplo anterior, o usuário ao clicar um botão no formulário, faz o computador gerar um único número aleatório, que dependendo do valor, apresenta resultados diferentes do nível do reservatório. O problema é que o usuário deverá clicar no botão toda vez que quiser gerar um valor. Muitas vezes pode ser interessante que o computador gere vários números aleatórios a fim de analisar um processo. Para isto usa-se o timer do Windows.

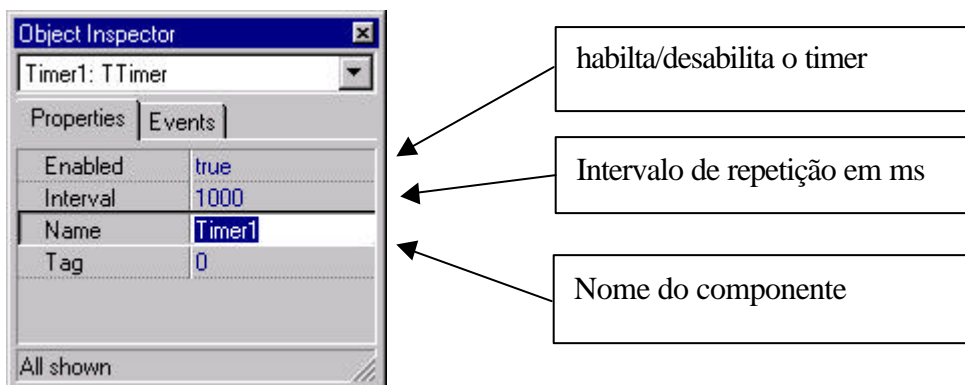
### 2.14.1. O Componente timer no C++ Builder

O componente timer pode ser acessado na barra de componentes dentro do menu system.



ícone do timer.

### 2.14.2. As propriedades do timer



### 2.14.3. Exemplo

Pode-se modificar o exemplo 3.13.2, para que os valores aleatório sejam gerados a cada 1s.

```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
    int x;
    x = random(10);
    switch(x)
    {
        case 0 :{
            Edit1->Text = "Reservatório vazio";
            break;
        }
        case 9 :{
            Edit1->Text = "Reservatório cheio";
            break;
        }
        default: Edit1->Text = "Reservatório Ok";
    }
}
```

Os comandos ficam dentro do evento timer.

### 2.14.4. Exercício

- 1- Desenvolva um programa que simule o sorteio da mega-sena (6 números de 0 a 60).

### 3. ESTRUTURAS HOMOGÊNEAS DE DADOS

---

#### **3.1. MATRIZES UNIDIMENSIONAIS (VETORES)**

Um vetor é um caso especial de matriz, onde ela possui apenas uma linha, e é declarado da seguinte forma:

*tipo nome[ tamanho ];*

Onde:

*tipo* especifica o tipo de dados que a matriz contém.

*nome* identifica a matriz dentro do programa;

*tamanho* valor constante que define a área de memória ocupada pelos elementos;

##### **3.1.1. Exemplos**

```
int vetor[10] = {1,2,3,4,5,6,7,8,9,10}; //inicializado com valores inteiros
```

```
float S[3] = {0,0,0}; //inicializado com zeros
```

```
long V[250]; // não inicializado
```

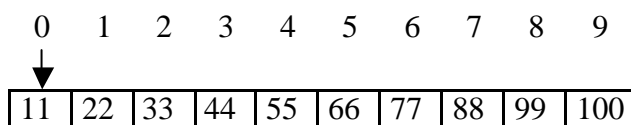
##### **3.1.2. Indexação**

Cada valor é armazenado em uma das posições do vetor (definido pelo tamanho). Então, o vetor possui índices que identificam a posição do conteúdo no vetor. O primeiro elemento sempre está na posição 0 (zero) do vetor.

##### **Exemplo**

```
int vetor[10] = {1,2,3,4,5,6,7,8,9,10};
```

pode ser representado na memória como:



Ou seja, um vetor contém um índice e um conteúdo armazenado numa posição indicada pelo índice.

O vetor V[0] contém o valor 11;

O vetor V[1] contém o valor 22;

...

O vetor V[9] contém o valor 100;

### 3.1.3. Exemplo

Para uma turma de 5 alunos, com notas iguais a 8.5, 9.0, 7.0, 7.5 e 9.5, escreva um programa que calcule a média da turma. As notas devem estar alocadas em um vetor.

```

#include <stdio.h>
#define ALUNOS 5
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    double notas[ALUNOS]={8.5,9.0,7.0,7.5,9.5};
    int pos = 0; // índices para os elementos da matriz
    double media = 0; // guarda o valor da média
    double soma = 0; // acumula o somatório das notas

    for (pos=0; pos <ALUNOS; pos++)
    {
        soma = soma + notas[pos]; // guarda a soma das notas
    }
    media = soma/ALUNOS; // calcula e guarda a média

    Edit1->Text = media; // mostra resultado
}

```



**3.1.4. Exercício**

- 1 - Modifique o exemplo acima para que o usuário forneça o valor das notas;
- 2 - Modifique o exemplo acima para que as notas sejam fornecidas aleatoriamente pelo programa;
- 3 - Altere o programa do item acima para classificar os elementos do vetor em ordem crescente e decrescente.

**3.2. Ordenação de vetores****3.2.1. Algoritmo de ordenação (bolha)****i) Troca de posição no vetor**

```

se vetor[x] > vetor[x+1] // se valor na posição x é
então // maior que o valor em x+1
    aux <- vet[x+1];
    vet[x+1] <- vet[x];
    vet[x] <- aux;
fim se

```

**ii) Número de repetições necessárias**

```

para vezes de 1 até N-1 faça // número de iterações
    para x de 1 até N-1 faça// para os N-1 elementos
        se vetor[x] > vetor[x+1] // realiza a troca
            então
                aux <- vet[x+1];
                vet[x+1] <- vet[x];
                vet[x] <- aux;
            fim se
        fim para
    fim para

```

**iii) Teste de mesa****Exemplo**

Dado o vetor  $\text{vet}[4] = \{50,36,1,8\}$ , ordenar os elementos em ordem crescente:

**primeira vez**

comparar  $\text{vet}[0]$  com  $\text{vet}[1]$  (50 com 36);

é maior, então troca;

comparar  $\text{vet}[1]$  com  $\text{vet}[2]$  (50 com 1);

é maior então troca;

comparar  $\text{vet}[2]$  com  $\text{vet}[3]$  (50 com 8);

é maior então troca.

O vetor  $\text{vet}$  agora contém  $\{36, 1, 8, 50\}$

**segunda vez**

comparar  $\text{vet}[0]$  com  $\text{vet}[1]$  (36 com 1);

é maior então troca

comparar  $\text{vet}[1]$  com  $\text{vet}[2]$  (36 com 8);

é maior então troca

comparar  $\text{vet}[2]$  com  $\text{vet}[3]$  (36 com 50)

não é maior então não troca

**após N-1 vezes**, o vetor  $\text{vet}$  contém  $\{1,8,36,50\}$ .

**3.2.2. Exercício**

1 - Escreva um programa em C, para preencher um vetor V de 6 elementos com valores aleatórios entre 0 e 10, e que ordene este vetor da seguinte forma:

a) botão 1: ordena automaticamente todos os elementos através do método bolha;

b) botão 2: ordena todos os elementos através do método bolha, mostrando passo a passo cada iteração (a cada clique do botão).

Obs.: crie um botão para gerar os números aleatórios para o vetor.

2 - Escreva um programa em C que preencha um vetor V de 100 elementos com valores aleatórios entre 0 e 10 e localize neste vetor, um valor procurado pelo usuário. Se o elemento existir neste vetor, apresentar uma mensagem que o elemento foi encontrado, caso contrário apresentar uma mensagem que o elemento não foi encontrado.

3 – Implemente modificações no método bolha para se diminuir a quantidade de comparações necessárias na ordenação, tornando-o mais eficiente.

4 - Pesquisar sobre o tema "Pesquisa Binária" e desenvolver o programa;

### 3.3. STRINGS

A string em C define uma sequência de caracteres alfanuméricos (texto). O exemplo: "CEP80215-901", contém letras, números e símbolos. Tudo o que está **entre aspas duplas** é dito string.

Uma string pode ser lida num formulário através de uma caixa de texto (EditBox), e armazenada em uma variável do tipo **char** (caracter) da seguinte forma:

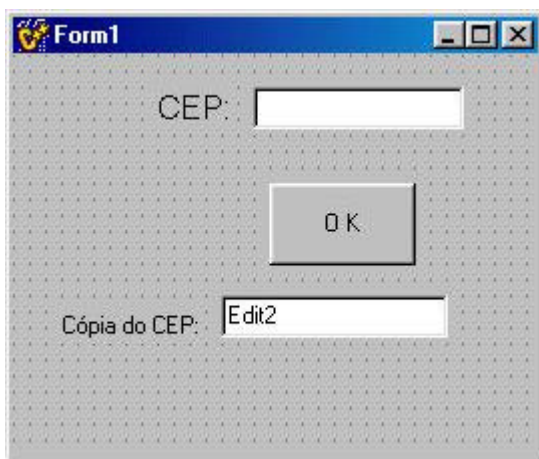
#### 3.3.1.Exemplo 1

Lembre que no C, 1 caracter ocupa 1 byte na memória . Então se definirmos uma variável do tipo char, estaremos reservando apenas 1 byte, possibilitando armazenar apenas um único caracter.

```
{
  char letra; // variável para armazenar um caracter.
  ...
  letra = 'a'; // a variável letra recebe o
               // caracter a, observe o uso de
               // aspas simples
  ...
  Edit->Text = letra;
}
```

### 3.3.2.Exemplo 2

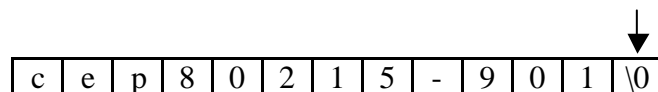
A string "CEP80215-901" ocupa 12 bytes. Como armazenar esta string, se a variável do tipo **char** possui tamanho de 1 byte? Se 1 caracter = 1 byte na memória, então para armazenar 12 caracteres são necessários 12 bytes. Então pode-se criar vetores para armazenar strings (texto).



No Botão OK:

```
{
  char cep[13];
  strcpy(cep, Edit1-Text.c_str());
  Edit2->Text = cep;
}
```

Um detalhe importante é lembrar que o C, na manipulação de strings, precisa de um byte a mais além do necessário, devido ao caracter de final de arquivo `\0` (barra zero). Se a string cep possui 12 caracteres, então deve ser criado o vetor com o tamanho 13, como apresentado abaixo.



Note o uso da função **strcpy()** para a leitura do Edit1, em substituição aos `atoi()`, `atof()`, etc.

### 3.3.3. Copiando strings

Quando se opera com strings faz-se necessário o uso de um comando específico para copiar a string, como apresentado abaixo:

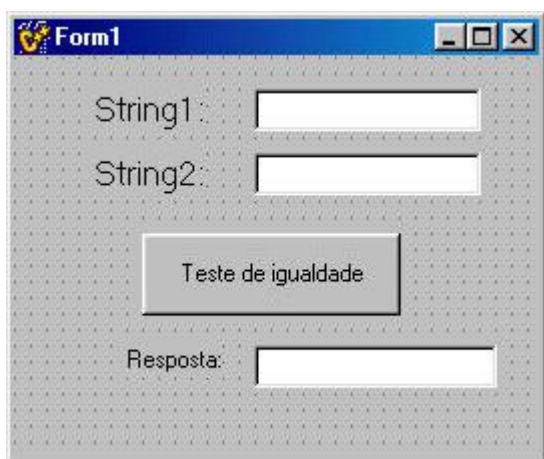
```
strcpy(string_destino, string_fonte)
```

### 3.3.4. Comparação de Strings

Muitas vezes é necessário realizar a comparação entre duas strings; mas elas não podem ser comparadas diretamente através dos operadores de comparação (símbolo `==`).

A função do C, chamada `strcmp(string1, string2)` compara duas strings, e retorna o resultado da comparação, através de um valor. Se este valor for 0 (zero), as duas strings são iguais, caso contrário são diferentes.

#### Exemplo



No botão [Teste de igualdade]:

```
{
    int x;
    char cep1[10];
    char cep2[10];
    strcpy(cep1,Edit1->Text.c_str());
    strcpy(cep2,Edit2->Text.c_str());
    x = strcmp(cep1,cep2);
    if (x==0) Edit3->Text = "iguais";
    else Edit3->Text = "diferentes";
}
```

### 3.3.5. Tamanho de strings

É possível comparar o tamanho de duas strings usando a função `strlen()`, como mostra o exemplo a seguir.

#### Exemplo

O **form** do exemplo anterior pode ser usado para testar qual string é a maior, entre os dois conteúdos fornecidos através das caixas de texto `Edit1` e `Edit2`:

```

{
    int x, y;
    char textoA[20];
    char textoB[20];
    strcpy(textoA, Edit1->Text.c_strs());
    strcpy(textoB, Edit2->Text.c_strs());
    x = strlen(textoA);
    y = strlen(textoB);
    if (x > y) Edit3->Text = "texto do Edit1 > texto do Edit2";
    if (x < y) Edit3->Text = "texto do Edit1 < texto do Edit2";
    if (x == y) Edit3->Text = "texto do Edit1 = texto do Edit2";
}

```

### 3.3.6. Comparação de Elementos da String

Para comparar elementos individuais da string, basta acessá-lo através de seu índice. Lembre que cada posição do vetor contém um único valor. Isto é ilustrado pelo exemplo a seguir:

```

{
    ... // se o primeiro caracter da string cep for igual a 8 ...
    if (cep[0] == '8') Edit3->Text = "Curitiba";
    else Edit3->Text = "Cep não cadastrado";
    ...
}

```

Então, para a comparação de elementos da string com valores do tipo **char**, pode-se utilizar diretamente os operadores relacionais com um comando **if/else** ou **switch/case**. Como um caracter da string, sozinho não é uma string, então não utiliza-se comandos de manipulação de string. Ou seja, por exemplo, não pode-se usar o comando `strcpy()` para copiar caracteres, e sim apenas quando deseja-se copiar toda a string.

### 3.3.7. Conversão de tipos

Muitas vezes pode ser necessário realizar alteração de tipos. Por exemplo **int** para **char**, etc.

### 3.3.7.1. convertendo valores numéricos para caracter

a) comando itoa(valor inteiro, string, base)

```
{  
    int num = 12345;  
    char string[6];  
    itoa(num, string, 10);  
    Edit1->Text = string;  
}
```

b) comando sprintf( string, format , argumentos);

```
{  
    int num = 12345;  
    char string[6];  
    sprintf(string,"%d", num);  
    Edit1->Text = string;  
}
```

```
{  
    float num = 123.456;  
    char string[20];  
    sprintf(string,"O valor é %.2f", num);  
    Edit1->Text = string;  
}
```

### 3.3.7.2. convertendo string para valores numéricos

a) comando atoi(string)

```
{  
    int num;  
    char string[6]="12345";  
    num = atoi(string);  
    num+=1;  
    Edit1->Text = num;  
}
```

b) comando `atof(string)`

```
{  
    float num;  
    char string[7]="1.2345";  
    num = atof(string);  
    num+=1;  
    Edit1->Text = num;  
}
```

### 3.3.8 Exercícios

- 1 – Num programa em C, o que acontece se a quantidade de caracteres digitada pelo usuário é maior que o tamanho da string definida no programa?
- 2 - O acontece de manipularmos uma string sem o '\0' ?
- 3 - Escreva um programa, que leia o valor de um número de CEP fornecido pelo usuário, e diga a qual cidade pertence este CEP; após identificar a cidade, fornecer o nome da rua associado a ele. O problema deve reconhecer no mínimo 3 cidades e 2 ruas cada. (cidades e ruas podem ser fictícios); O primeiro caracter da string do cep se refere à cidade; os restantes correspondem à rua; O programa deve identificar se o número fornecido pelo usuário é válido, através dos requisitos: tamanho do cep, presença do sinal "-", presença de uma cidade cadastrada e presença de uma rua cadastrada.
- 4 – Digite um programa que coloque em ordem alfabética (crescente) um string de 20 caracteres.
- 5 – Digite um programa para inverter uma string fornecida pelo usuário.



### 3.4. MATRIZES

Uma Matriz é uma coleção de variáveis do mesmo tipo, referenciada por um nome comum. Um elemento específico de uma matriz é acessado através de um índice. Todos os elementos ocupam posições contíguas na memória.

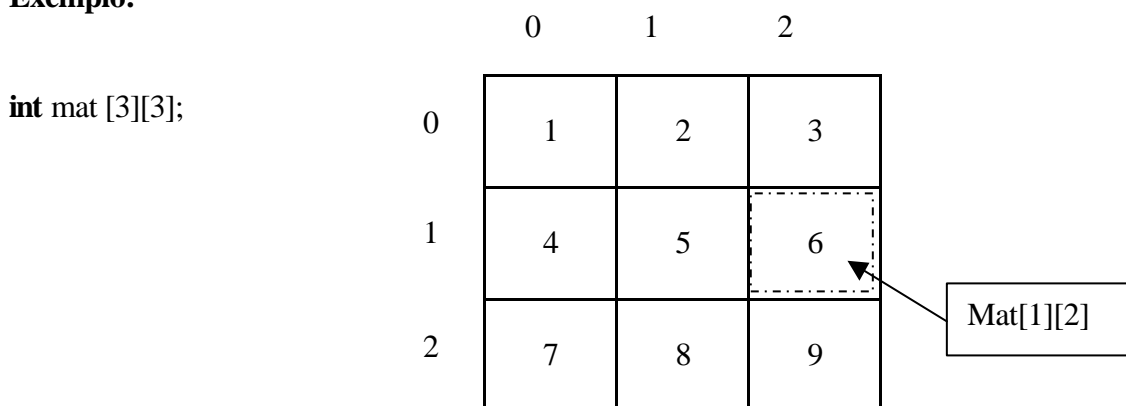
#### 3.4.1. Matrizes Bidimensionais

As matrizes em C podem ser de várias dimensões. Uma matriz de duas dimensões pode ser declarada da seguinte forma:

**int nome[tamanho 1] [tamanho 2];**

Os dados de uma matriz são acessados através de índices. Existe um índice associado a uma linha da matriz e outro índice para a coluna da matriz, de onde está alocado o dado.

**Exemplo:**



As posições da matriz são formadas por:

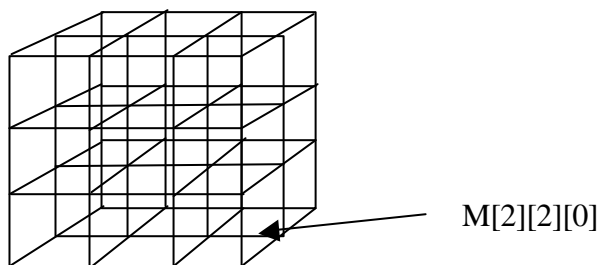
mat[0][0] contém 1	mat[1][0] contém 4	mat[2][0] contém 7
mat[0][1] contém 2	mat[1][1] contém 5	mat[1][0] contém 8
mat[0][2] contém 3	mat[1][2] contém 6	mat[2][0] contém 9

### 3.4.2. Matrizes multidimensionais

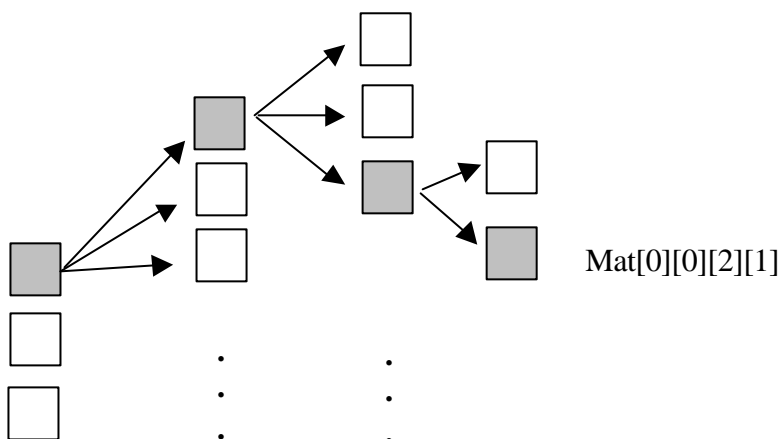
Uma matriz pode ter mais de uma dimensão, como apresentado pelos exemplos a seguir

#### **Exemplo**

a) `int M[3][3][2]`



a) `int Mat[3][3][3][2]`



### 3.4.3. Matrizes de strings

Uma matriz do tipo **char**, é um caso especial na manipulação de matrizes, porque cada linha da matriz, é uma string. Se cada linha da matriz é uma string, a linha pode ser manipulada com os comandos conhecidos de string.

Cada linha pode ser acessada somente pelo índice da linha. Os caracteres individuais também pode ser acessados através dos índices das posições, veja o exemplo:

**Exemplo**

```

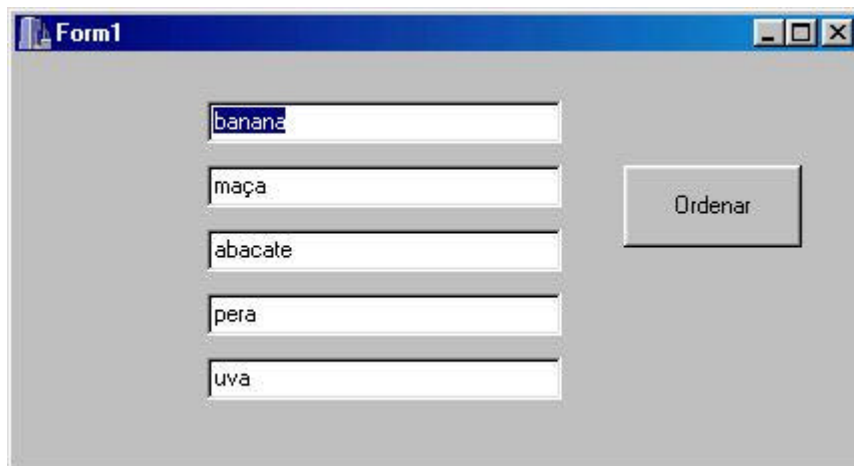
char M[5][8] = {"banana",
               "maça",
               "abacate",
               "pera",
               "uva"}; // matriz M global

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Edit1->Text = M[0];
    Edit2->Text = M[1];
    Edit3->Text = M[2];
    Edit4->Text = M[3];
    Edit5->Text = M[4];
}

```

Note que é necessário apenas o índice da linha. Isto somente para matrizes do tipo **char**.

No form:



Observe que na definição da matriz M do exemplo, foi reservado espaço para 5 linhas de 8 caracteres (colunas) suficiente para armazenar os nomes das frutas mais o '\0'.

Para acessar um caractere individual, basta informar os índices da linha e coluna, como no exemplo abaixo:

```
Edit1->Text = M[0][0]; // apresentará no Edit1 a letra 'b'
```

**3.4.4. Exercícios**

- 1) Escreva um programa em C que preencha uma matriz 4X4 com "0" e a sua diagonal principal com "1", e apresente a matriz através de um **form**;
- 2) Para um professor que tenha 3 turmas de 5 alunos, escreva um programa que calcule a média de notas de cada turma;
- 3) Escreva um programa em C que preencha uma matriz 4X4 com "0" e a sua diagonal principal com "1", e apresente na tela uma linha da matriz por vez, a cada clique em um botão;
- 4) Desenvolva um programa do “jogo da velha”, de forma que o usuário possa jogar contra o computador;
- 5) Desenvolva um programa do tipo “tetris” usando matriz e os conceitos de C já adquiridos;
- 6) Desenvolva um programa que permita obter a transposta de uma matriz 4 x 4 fornecida pelo usuário;
- 7) Desenvolva um programa que permita somar, subtrair e multiplicar duas matrizes 3 x 3, fornecidas pelo usuário;
- 8) Desenvolva um programa que some todos os elementos de uma matriz 5 x 5;
- 9) **Desenvolva um programa que permita movimentar o elemento 1 da matriz abaixo em uma direção aleatória a cada 1s. O movimento do elemento não pode extrapolar os limites da matriz.**

	1	

## 4. PONTEIROS EM C

---

### 4.1. DEFINIÇÃO

Um ponteiro é uma variável que contém um endereço de memória [Schildt,97]. Este endereço é normalmente a posição de outra variável na memória. Se uma variável contém o endereço da outra, então a primeira variável aponta para a segunda.

#### Exemplo

Endereço na Memória	Conteúdo na Memória
1000	
1001	1100
...	
...	
1100	'A'
1101	

Neste exemplo, uma variável que está alocada na posição 1001 possui como conteúdo, o valor 1100; - este valor é o endereço de uma outra posição de memória, que possui uma informação armazenada, por exemplo, o caracter 'A'. Então neste exemplo, diz-se que a variável da posição 1001 aponta para a variável da posição 1100, e ambas tem o mesmo conteúdo, a letra 'A'

No C, ao se definir:

```
char letra; // cria variável do tipo char
...
letra = 'A'; // a variável letra armazena o caracter A
...
```

É reservada uma área na memória com o tamanho especificado pelo tipo da variável (char = 1 byte). Esta área é identificada no programa por um nome (letra), mas internamente ao sistema, por um valor numérico, chamado endereço da variável. O conteúdo da variável também é convertido para um número na base binária.

O valor do endereço de memória não é escolhido pelo programador. O programa, em execução, é que se encarrega de achar uma posição de memória livre no computador, e a reserva para uso. Entretanto é possível e desejável, conhecer estes endereços. Veja a seguir a declaração de uma variável tipo ponteiro.

## 4.2. Declaração de um ponteiro

Para declarar e utilizar um ponteiro usa-se uma simbologia apropriada. O símbolo **&** (e-comercial) se refere ao endereço de uma variável. O símbolo **\*** (asterisco) declara um ponteiro, e também serve para referenciar um conteúdo. A declaração é dada por:

**tipo \* nome\_da\_variável;**

Assim, alterando a definição da variável letra do Exemplo do item 4.1 acima, para o uso de ponteiro, teremos (as linhas foram numeradas para explicar as linhas do trecho de programa):

### Exemplo

```
[1] char letra; // cria variável do tipo char
[2] char * pLetra; // variável do tipo ponteiro para char
...
[3] letra = 'A'; // a variável letra armazena o caracter A
...
[4] pLetra = & letra; // pLetra aponta para letra
```

Na linha [1] criou-se uma variável do tipo char para armazenar uma letra.

Na linha [2] está sendo declarado um ponteiro chamado pLetra do mesmo tipo de letra (char). Sempre um ponteiro, deve ser do mesmo tipo da variável a ser apontada. O símbolo **\*** (asterisco) indica que a variável é um ponteiro.

Na linha [3] a variável letra (uma posição da memória) recebe o caracter 'A', ou seja, o conteúdo a ser armazenado na área reservada para variável letra.

Na linha [4] o ponteiro pLetra recebe o endereço de memória da variável letra, ou seja, passa a apontar para a variável letra. Usa-se o símbolo & (e-comercial) para acessar o endereço de uma variável.

Se adicionarmos uma nova variável, aux, junto às definições de variáveis do exemplo:

```
char aux; // variável auxiliar do exemplo
```

e, acrescentarmos a linha [5]:

```
[5] aux = *pLetra; // aux recebe o conteúdo de pLetra
```

teremos o seguinte:

Na linha [5] a variável aux recebe o conteúdo da posição de memória apontada por pLetra, ou seja 'A'. Em termos práticos, aux e letra acessam a mesma posição de memória. O símbolo \* (asterisco) agora representa o "conteúdo de ..."

### 4.3. Exemplos

1. Dado o seguinte trecho de programa, qual o conteúdo das variáveis após a sua execução?

```
void procl(void)
{
  int x = 1;
  int y = 2;
  int *ip; // declaração de um ponteiro para inteiro
  ip = &x; // o ponteiro ip recebe o endereço de x
  y = *ip; // y recebe o conteúdo de x
  *ip = 0; // x é zero
}
```

**Resposta**

endereço	conteúdo	endereço	conteúdo	endereço	conteúdo	endereço	conteúdo
1001		1001		1001		1001	
1002	1	1002	1	1002	1	1002	0
1003	2	1003	2	1003	1	1003	1
1004		1004	1002	1004	1002	1004	1002
1005		1005		1005		1005	
1006		1006		1006		1006	

- 1.o) `int x = 1;`  
`int y = 2;`
- 2.o) `ip = &x;` ip só é adicionado na memória após ser inicializado
- 3.o) `y = *ip;`
- 4.o) `*ip = 0;`

2. Qual é o valor de x e de y, após a execução do procedimento troca() das situações A e B?

Situação A	Situação B
<pre> // globais int x = 0; int y = 1; // procedimento troca void troca(int val_x, int val_y) {     int temp;     temp = val_x;     val_x = val_y;     val_y = temp; } //dentro de um botão: {     troca(x,y); } </pre>	<pre> // globais int x = 0; int y = 1; // procedimento troca void troca(int * val_x, int * val_y) {     int temp;     temp = *val_x;     *val_x = *val_y;     *val_y = temp; } //dentro de um botão: {     troca(&amp;x,&amp;y); } </pre>



#### 4.4. Ponteiros para matriz

Dados do tipo matriz podem ser indexados através de ponteiros, como mostram os exemplos a seguir:

##### Exemplo 1

```
{
  int v[10];
  int *p; // o sinal * indica que a variável é do tipo ponteiro
  p = v; //atribui a p o endereço do primeiro elemento de vetor
}
```

##### Exemplo 2

```
{
  int i[10];
  int *p; // o sinal * indica que a variável é do tipo ponteiro
  p = i; // atribui a "p" o endereço do primeiro elemento de i
  p[5] = 100; // 6º elemento recebe 100 usando o índice
  *(p+5) = 100; // 6º elemento recebe 100
}
```

Dado o vetor `i[10]` definido como expresso a seguir:

i[0]	i[1]								i[9]
------	------	--	--	--	--	--	--	--	------

Se declaramos um ponteiro para inteiro:

```
int *pont;
```

a atribuição:

```
pont = i;
```

faz com que "pont" aponte para o elemento zero do vetor "i" (`&i[0]`);

Se "pont" aponta para um elemento particular de um vetor, então "pont+1" aponta para o próximo elemento.

Ou seja:

`pont + n` aponta para `n` elementos após `pont`.

`pont - n` aponta para `n` elementos antes de `pont`.

**Obs.1:** O nome do vetor já é o endereço do vetor;

**Obs.2:** Também são válidas as operações com ponteiros para matrizes multidimensionais.

### Exemplo

`mat[10][10]`; equivale a `&mat[0][0]`

`mat[0][4]`, pode ser acessado por `*(mat+3)`

`mat[1][2]`, pode ser acessado por `*(mat+13)`

### Exemplo 3

Elabore uma função em C que retorne o tamanho de uma cadeia de caracteres, usando ponteiros.

```
int strlen(char *S)
{
    int n;
    for ( n = 0; *S != '\0' ; S++ ) n++;
    return n;
}
```

## 4.5. Vetores de Ponteiros

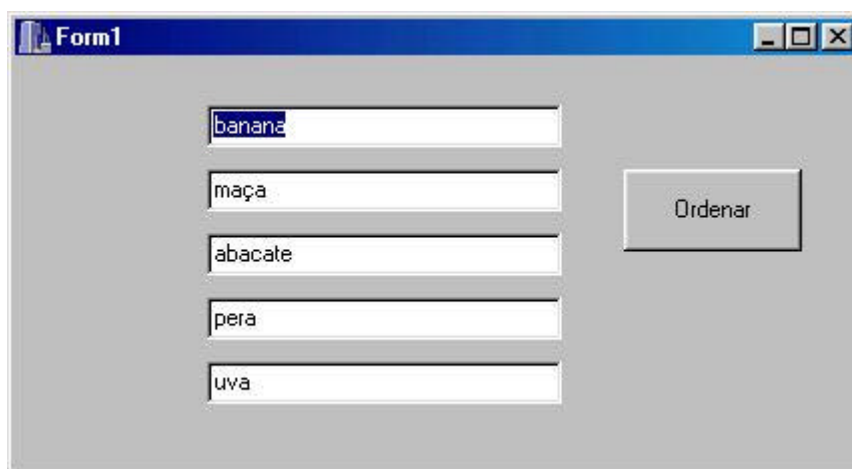
Da mesma forma que se cria uma variável ponteiro, pode-se criar também um vetor de ponteiros. O vetor de ponteiros é útil quando se trabalha com informações de tamanho diferente. Por exemplo, para ordenar um vetor de inteiros, pode-se trocar os elementos porque todos têm o mesmo comprimento em bytes. Mas, para ordenar texto, é necessário mais do que uma única operação de comparação. Um vetor de ponteiros é dado através da seguinte definição:

**tipo** \*nome\_ponteiro[tamanho];

### 4.5.1. Exemplo 1

Escreva um programa que coloque em ordem alfabética as frutas armazenadas na string M, apresentada abaixo. Este exemplo não usa ponteiros.

### Resposta



a) Definição das variáveis globais:

```
#define comp 8 // matriz M global
char M[5][comp] = {"banana",
                  "maça",
                  "abacate",
                  "pêra",
                  "uva"}; // matriz M global
```

b) Inicializado os EditText na criação do Form1:

```
//-----
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    Edit1->Text = M[0]; // atualiza os EditText na criação do formulário.
    Edit2->Text = M[1]; // cada linha da matriz,
    Edit3->Text = M[2]; // pode ser acessada como se
    Edit4->Text = M[3]; // fosse uma string
    Edit5->Text = M[4];
}
```

c) Linhas de programa associadas ao botão "ordena":

```
void __fastcall TForm1::OrdenaClick(TObject *Sender)
{
    int nvezes, x;
    char aux[comp]="";
    for (nvezes = 0; nvezes < 4; nvezes++)
    {
        for (x=0;x<=3;x++)
        {
            if (M[x][0] > M[x+1][0]) // testa o primeiro elemento de cada string
            {
                strcpy(aux,M[x]); // usa o método "bolha" para ordenar
                strcpy(M[x],M[x+1]);
                strcpy(M[x+1],aux);
            }
        }
    }
    Edit1->Text = M[0];
    Edit2->Text = M[1];
    Edit3->Text = M[2];
    Edit4->Text = M[3];
    Edit5->Text = M[4];
}
```

**4.5.2.****Exercício**

Modifique o exemplo 4.5.1, para que a atualização dos EditText sejam feitos através de um procedimento.

**4.5.3. Exemplo 2**


Modifique o exemplo 4.5.1, para usar ponteiros na ordenação da matriz de frutas M.

**Resposta**

a) Variáveis Globais:

```
#define comp 8
#define linhas 5
char M[linhas][comp] = {"banana",
                        "maça",
                        "abacate",
                        "pera",
                        "uva"};

char * pM[linhas];
```



b) Linhas de programa associadas ao botão "ordena":

```
{
  int nvezes, x;
  char *aux;
  int fim = 3;
  for (x=0; x<5; pM[x] = M[x],x++); // apontar para elementos de M
  for (nvezes = 0; nvezes < 4; nvezes++)
  {
    for (x=0; x<=fim; x++)
    {
      if (*pM[x] > *pM[x+1]) //[1]
```

```

{
    aux    = pM[x];
    pM[x]  = pM[x+1];
    pM[x+1]= aux;
    fim--; [2]
}
}
}
Edit1->Text = pM[0]; //[3]
Edit2->Text = pM[1];
Edit3->Text = pM[2];
Edit4->Text = pM[3];
Edit5->Text = pM[4];
}

```

Observe que:

- As linhas de M apresentadas nos Edits, ainda seguem a mesma ordem (pM[0] até pM[4]). é apresentado;
- Agora, os ponteiros apontam para as linhas de M. Note que as posições de M não são alteradas.

#### **4.5.4. Exercícios**

- 1 - O que acontece se a linha [1] for modificada para **if (pM[x] > pM[x+1]) ...**
- 2 - Para que serve a linha [2]?
- 3 - O que acontece se as atribuições aos Edits a partir da linha [3] forem modificadas para **Edit1->Text = \*pM[0]; ... Edit1->Text = \*pM[4];**
- 4 – Desenvolva o programa de busca binária (aula de vetores), agora usando ponteiros.
- 5 – Desenvolva uma função para somar o conteúdo de duas matrizes 3x3 usando ponteiros.

## 5. ALOCAÇÃO DINÂMICA DE MEMÓRIA

---

### 5.1. Introdução

Existem duas maneiras de um programa em C armazenar informações na memória. A primeira é através da utilização de variáveis locais e globais. Como visto anteriormente, uma variável global, existe na memória enquanto o programa estiver sendo executado e pode ser acessada ou modificada por qualquer parte do programa. Já, a variável local é acessível, apenas dentro de um procedimento ou função (ou dentro de um componente de formulário), ou seja, criada e usada no momento em que for necessária.

A segunda maneira de armazenar informações na memória é utilizar alocação dinâmica. Desta forma o programa usa espaços da memória que podem variar de tamanho de acordo com o que se precisa armazenar. Isto ocorre em tempo de execução do programa.

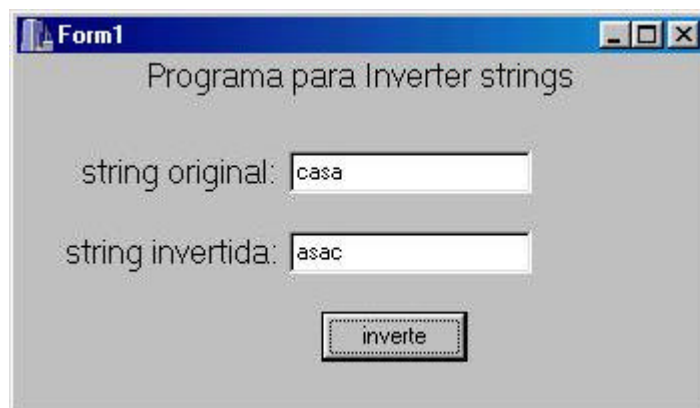
### 5.2. Comando de alocação

Um dos comandos que realiza a alocação dinâmica no C é o **malloc()**, que significa *memory allocation*. A função `malloc()` devolve um ponteiro para o primeiro byte de uma região de memória de tamanho **size**. Caso não haja memória suficiente, `malloc` devolve um ponteiro NULL (nulo), ou seja, sempre deve ser verificado se não é devolvido um ponteiro nulo antes de usar a memória requisitada.

#### 5.2.1. Exemplo de Alocação Usando o Comando `malloc()`

Para usar os comandos de alocação é necessário incluir `<alloc.h>`. O exemplo a seguir cria um programa para apresentar na ordem inversa, uma string fornecida por um usuário.

#### **Exemplo**



a) Dentro do botão "inverte", o trecho do programa que realiza a alocação:

```
void __fastcall TForm1::InverteClick(TObject *Sender)
{
    char *s;
    char aux[2];
    char buf[41]="";
    int x=0;
    s = (char *) malloc(40);
    if (!s)
    {
        Application->MessageBox("mensagem", "erro de alocação", MB_OK);
        exit(1);
    }

    // continua....
}
```

Na linha de código **s = (char \*) malloc(40);** do exemplo acima, foi criado um ponteiro "s" do tipo char. Este ponteiro vai apontar para o endereço inicial, da região de memória alocada por malloc(). Note que o comando malloc recebe como parâmetro, o valor da quantidade de memória requerida, neste caso é 40. Mas o que significa este valor "40"? Este tamanho reserva na memória 40 espaços de tamanho (**char \***), ou seja, 40 bytes, que estão reservados pelo programa e podem, agora, ser acessados através do ponteiro "s".

Após a criação de "s", é necessário (e obrigatório) testar se o ponteiro existe, ou seja, se ele aponta para algum endereço na memória não nulo. Se por algum motivo, o ponteiro não puder



ser criado, então ele conterá um valor nulo (NULL ou 0). Se isto ocorrer, a execução do programa deve ser terminada, sob pena de tornar instáveis os outros aplicativos em execução, ou até travar o sistema operacional. O comando **if (!s)** testa se o ponteiro existe, ou seja, se contém um valor diferente de nulo. Se o ponteiro for nulo, é apresentada uma mensagem de erro e o programa deve ser encerrado. O comando **exit(1)** encerra o programa neste ponto.

As outras variáveis locais, definidas no exemplo acima, serão apresentadas a seguir.

```
// continuação....

strcpy(s, Edit1->Text.c_str()); [1]
for(x = strlen(s)-1; x >= 0; x--) [2]
{
    sprintf(aux, "%c", s[x]); [3]
    strcat(buf, aux); [4]
}
Edit2->Text = buf; [5]
free(s); [6]
```

Seguindo o exemplo, na linha:

- [1] O conteúdo do EditBox é passado para a região de memória apontada por "s".
- [2] É executado um loop que varia de x (tamanho de s) até 0, que é a primeira posição da string s (s[0]).
- [3] Converte um caracter em uma string .
- [4] A variável local buf concatena cada valor de aux.
- [5] Buf é apresentado no formulário;

Após a utilização da variável alocada dinamicamente, é extremamente importante liberar a área de memória alocada por malloc, usando o comando **free()**. A linha

- [6] libera a área de memória apontada pelo ponteiro "s".

### 5.2.2. Melhorando o Uso de Ponteiros

Pode-se melhorar o programa, realizando algumas modificações no uso de ponteiros, como mostra o exemplo abaixo:

```

Void __fastcall TForm1::InverteClick(TObject *Sender)
{
    char *s; [1]
    char *aux; [2]
    int x=0, y=0; [3]

    s = (Edit1->Text.c_str()); [4]

    while (*s) { s++; x++; } [5] // calcula o tamanho da string

    aux = (char *) malloc(x); [6] // cria dinamicamente uma área para aux
    if (!aux) [7]
    {
        Application->MessageBox("mensagem","erro de alocação",MB_OK);
        exit(1); [8]
    }

    while(y < x) {aux[y++] = *(--s); } [9] // inverte a string
    aux[y] = '\0'; [10]
    Edit2->Text = aux; [11]
    free(aux); [12]
}

```

Neste exemplo, temos as seguintes modificações:

- [1] e [2] tanto "s" quanto "aux" são ponteiros para string, "buf" não é mais necessário.
- [3] duas variáveis; "x" que armazenará o tamanho de "s", e "y" para indexar "aux".
- [4] "s" recebe o endereço da string de Edit1; lembre que uma string é um vetor de caracteres cujo primeiro elemento é o endereço para a string toda. Desta forma não precisa-se usar os comandos para manipular strings.

- [5] outra maneira de calcular o tamanho de uma string, usando ponteiros.
- [6] "aux" aponta para uma nova região na memória, do tamanho de "s".
- [7] testa se o ponteiro existe, ou seja, se "aux" aponta para um endereço diferente de nulo.
- [8] encerra o programa se a memória não foi alocada.
- [9] inverte o conteúdo da string "s" e armazena em "aux". Incrementa-se índice de y e move o ponteiro de "s" para o início de "s".
- [10] acrescenta finalizador de string.
- [11] Edit2 recebe aux; na verdade, o Edit aponta para aux.
- [12] libera a área apontada por "aux".

Como pode ser observado, neste exemplo não se fez necessário o uso de bibliotecas de manipulação de string, ou seja, o programador tem maior controle sobre o que está sendo executado. Muitas vezes, linguagens de mais alto nível, mascaram efetivamente o que acontece após o comando ser executado. Da forma apresentada neste exemplo, o programador consegue entender o que está acontecendo durante a execução do programa, evitando inserir comandos desnecessários.

### **5.3. Exercícios**

- 1- O que acontece se for alocado para aux um tamanho de memória diferente do valor de x?
- 2- Crie um EditText no formulário para apresentar o conteúdo do ponteiro "aux", após ser executada a linha [12]. O que será apresentado na tela? Que cuidados deve-se ter ao liberar a memória apontada pelos ponteiros?

### **5.4. Portabilidade**

O tamanho em bytes, de um dado tipo de variável, pode mudar de máquina para máquina. Para evitar problemas com diferenças de tamanhos e assegurar a portabilidade de programas, usa-se o operador **sizeof()**. sizeof informa ao programa o tamanho cujo tipo de variável ocupa na memória.

### 5.4.1. Exemplo do uso de sizeof

```
{
  int *pont; [1]
  ...
  pont = (int *) malloc(sizeof(int)); [2]
  if (!pont) [3]
  {
    Application->MessageBox("mensagem","erro de alocação",MB_OK);
    exit(1); [4]
  }
  ...
}
```

do exemplo acima, observa-se em:

- [1] declaração de um ponteiro para inteiro chamado "pont";
- [2] "pont" aponta para o endereço da posição de memória definida pelo comando malloc; note que o tamanho a ser criado é especificado pelo sizeof(). Porque provavelmente, o inteiro usado neste exemplo deve reservar 4 bytes por estar operando sobre o Windows; se o mesmo programa fosse executado no DOS, provavelmente, seria reservado 2 bytes (Ansi C).
- [3] realiza o teste de validade do ponteiro.
- [4] se o ponteiro não for criado, aborta a execução do programa.

### 5.5. EXERCÍCIOS

1- Escreva um programa para ler N valores reais fornecidos por um usuário, e que calcule a média destes valores e apresente o resultado. O programa deve usar ponteiros.

#### **Resposta:**

a) Variáveis globais usadas no programa:

```
float *valores, soma, media;
int N, flag, vezes;
char msg[20];
```

b) Variáveis inicializadas na criação do formulário:

```
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    vezes = flag = soma = media = 0;
}
```

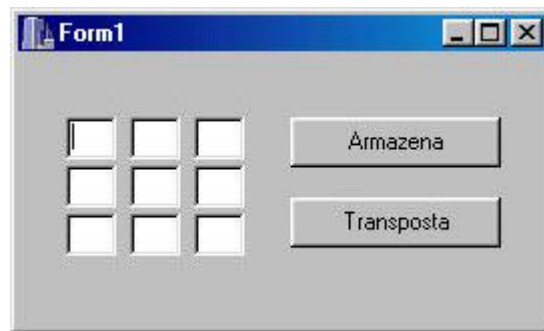
c) Comandos associados ao botão:

```
{
    if (!flag)// só entra para criar a área de memória
    {
        N = atoi(Edit1->Text.c_str());
        valores = (float *)malloc(N * sizeof(float));
        if (valores == NULL)
        { Application->MessageBox("mensagem","erro de alocação",MB_OK);
            exit(1);
        }
        flag = 1;
    }
    if (vezes < N)
    { *valores = atoi(Edit2->Text.c_str());
        soma += *valores;
        media = soma/N;
        valores++;
        vezes++;
    }
    else
    { free(valores);
        flag = vezes = N = 0;
        sprintf(msg,"média = %f",media);
        Edit2->Text = msg;
        Edit1->Clear();
    }
}
```

2- Responda as seguintes perguntas:

- O ponteiro "valores" do exercício acima é um vetor?
- Para que serve o comando valores++?
- O que acontece, se não for especificado um valor de N ao executar o programa?
- A variável **char** msg[20] pode ser declarada como **char \*msg**?

3- Escreva um programa que leia uma matriz fornecida pelo usuário, e apresente na tela a transposta desta matriz. Use alocação dinâmica para armazenar a matriz. E ponteiros para acessar seu conteúdo.



```
int *M; // ponteiro para uma matriz declarado como global

//-----
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    M = (int *)malloc(9 * sizeof(int));
    if (!M)
    {
        ShowMessage("falta memória");
        exit(0);
    }
}
```

```

//-----
void __fastcall TForm1::ArmazenaClick(TObject *Sender)
{
    *(M+0) = atoi(Edit1->Text.c_str());
    *(M+1) = atoi(Edit2->Text.c_str());
    *(M+2) = atoi(Edit3->Text.c_str());
    *(M+3) = atoi(Edit4->Text.c_str());
    *(M+4) = atoi(Edit5->Text.c_str());
    *(M+5) = atoi(Edit6->Text.c_str());
    *(M+6) = atoi(Edit7->Text.c_str());
    *(M+7) = atoi(Edit8->Text.c_str());
    *(M+8) = atoi(Edit9->Text.c_str());
}

//-----
void __fastcall TForm1::TranspostaClick(TObject *Sender)
{
    // ... inserir o código aqui.
}

//-----
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
    free(M);
}

//-----

```

4- Modifique o exercício 3, para que o espaço alocado dinamicamente para a matriz M, seja criado de acordo com a dimensão da matriz, especificada pelo usuário.

5- Modifique o exercício 4.3, criando uma função que aloca espaço dinamicamente, e que possa ser chamada sempre que for preciso alocar memória.

```

//-----
// função que realiza alocação dinâmica de memória de acordo com um tamanho
// especificado pelo usuário

int * aloca(int size)
{
    int *pMemoria;
    pMemoria = (int *)malloc(size);
    if (!pMemoria)
    {
        ShowMessage("falta memória");
        return(NULL);
    }
    return pMemoria;
}

```

Neste caso, tem-se uma função do tipo ponteiro para inteiro. Ou seja, o valor retornado é um ponteiro, cujo valor é NULL se a memória não puder ser alocada, ou um endereço de memória válido, para a região criada. O tamanho desta região é definida pela variável size, que tem o valor passado pelo usuário, como parâmetro para esta função. Para usar esta função, basta criar uma variável ponteiro para apontar para o endereço retornado pela função.

```

//-----
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    M = aloca(9 * sizeof(int));
}

```

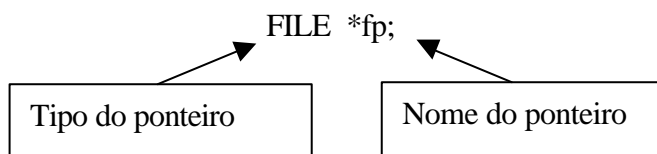


## 6. ARQUIVOS EM C

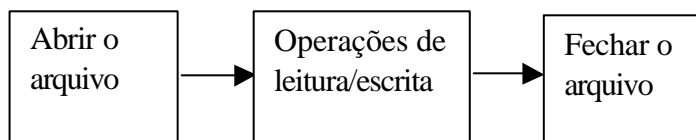
---

### 6.1. Ponteiro de arquivo

Arquivos são coleções de dados que recebem um nome único, pelo qual os dados podem ser acessados e manipulados. Um ponteiro de arquivo é uma variável ponteiro do tipo **FILE**, que é uma estrutura declarada em `<stdio.h>` e contém informações sobre o arquivo. Um ponteiro para arquivo pode ser obtido da seguinte forma:



Para operar com arquivos é necessário realizar as seguintes operações:



### 6.2. Abrindo arquivos

Antes de qualquer operação com arquivos, é necessário que este arquivo exista. Para isto, é preciso abrir um arquivo, que em C é dado pelo comando **fopen()**, da seguinte forma:

```
FILE * fopen (nomearquivo, modo)
```

Onde:

**nomearquivo** é uma string e pode incluir o caminho para o arquivo e a extensão.

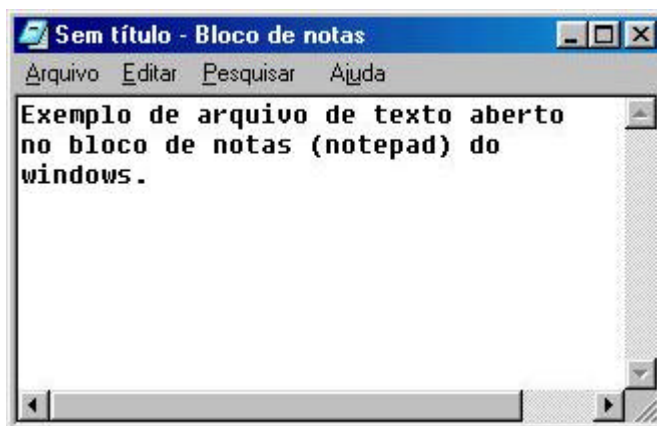
**modo** determina como o arquivo será aberto. O arquivo pode ser do formato `texto`, ou do formato binário. Os valores para modo são:

Modo	Descrição
<i>rt</i>	abre arquivo texto para leitura
<i>wt</i>	abre arquivo texto para escrita
<i>at</i>	anexa dados a um arquivo texto
<i>rb</i>	abre arquivo binário para leitura
<i>wb</i>	abre arquivo binário para escrita
<i>ab</i>	anexa elementos a um arquivo binário
<i>r+t</i>	abre um arquivo texto para leitura/escrita
<i>w+t</i>	cria um arquivo texto para leitura/escrita
<i>a+t</i>	anexa dados a um arquivo texto para leitura/escrita
<i>r+b</i>	abre um arquivo binário para leitura/escrita
<i>w+b</i>	cria um arquivo binário para leitura/escrita
<i>a+b</i>	anexa a um arquivo binário para leitura/escrita

### **6.2.1. Arquivos Tipo Texto**

Qualquer arquivo cujo conteúdo está expresso pelos valores de caracteres da tabela ASCII. Normalmente um arquivo texto não contém nenhum tipo de formatação, e pode ser aberto em qualquer aplicativo.

**Exemplo:**





### 6.3.1. Observações

- a) Se o arquivo "teste.txt" não existir, o comando `fopen()` cria este arquivo em branco. Se a opção for "w", ele grava as informações neste arquivo recém criado.
- b) Se o arquivo "teste.txt" já existir, o comando `fopen()` abre este arquivo. Se a opção for "w", ele grava as novas informações sobre o conteúdo antigo. Ou seja, as informações anteriores são perdidas. Para que isso não aconteça, deve-se usar o modo "a" para anexar dados ao arquivo existente.
- c) O que acontece se um arquivo não puder ser criado, como por exemplo, se não houver mais espaço em disco, ou se um disquete não estiver inserido no drive especificado? Então, antes de usar este arquivo, é necessário saber se o arquivo foi criado corretamente, para isso é obrigatório testar o conteúdo do ponteiro de arquivo. Assim, o comando para abrir um arquivo, pode ser escrito da seguinte forma:

```
{  
FILE *fp; [1]  
fp = fopen("teste.txt", "wt"); [2] if (fp == NULL) [3]  
{  
    exit(1);  
}  
}
```

Onde:

[1] é uma variável ponteiro do tipo FILE;

[2] abre o arquivo teste.txt tipo texto no modo de escrita (w);

[3] testa se o ponteiro é diferente ou igual a nulo, se for NULL interrompe a execução do programa.

#### **6.4. Abrindo um arquivo para leitura**

Da mesma forma que é necessário abrir (ou criar) um arquivo para escrever algo, pode-se abri-lo apenas para leitura das informações. Desta maneira o conteúdo do arquivo não pode ser alterado, e sim, apenas consultado. O comando para abrir um arquivo para leitura é dado como mostra o exemplo abaixo:

```
{  
FILE *fp; [1]  
fp = fopen("teste.txt","rt"); [2]  
if (fp == NULL) [3]  
{  
    exit(1);  
}  
}
```

Onde:

[1] é uma variável ponteiro do tipo FILE;

[2] abre o arquivo teste.txt tipo texto no modo de leitura (r);

[3] testa se o ponteiro é diferente ou igual a nulo, se for NULL interrompe a execução do programa.

#### **6.5. Fechando um arquivo**

O comando **fclose()** fecha um arquivo que foi aberto pela chamada de **fopen()**. Sempre, a operação com arquivos acontece nesta sequência:

Um arquivo aberto, quando não for sofrer mais nenhum tipo de operação, obrigatoriamente deve ser fechado. Se não for fechado, o arquivo pode ficar definitivamente inacessível.

## 6.6. Comandos de escrita e leitura

### 6.6.1. fputc()

A função `fputc()` escreve um caracter na sequência de texto (stream) na posição atual do arquivo, e avança o indicador de posição do arquivo. O valor retornado pela função é o valor do caracter escrito. Se houver um erro, o valor EOF (End Of File) é devolvido pela função. A sintaxe do comando é dada por:

```
fputc(int ch, FILE *pont);
```

Onde,

ch é o caracter a ser armazenado; (a função converte este int para unsigned char)

pont é o ponteiro para o arquivo.

**Exemplo 1:** Gravando um arquivo em modo texto.

```
{
FILE *fp; [1]

char *str = "texto de exemplo"; [2]

fp = fopen("c:/temp/fputc.txt", "wt"); [3]

if (fp == NULL) // [4]
{
    exit(1);
}

while(*str) if (fputc(*str++, fp) != EOF); [5]
fclose(fp); [6]
}
```

Onde:

[1] é uma variável ponteiro do tipo FILE;

[3] abre o arquivo fputc.txt do tipo texto no modo de escrita (w);

[4] testa se o ponteiro é diferente ou igual a nulo, se for NULL interrompe a execução do programa;

[5] enquanto o conteúdo da string apontada por str for diferente de '\0', cada caracter é passado para o arquivo apontado por fp, se não ocorrer uma mensagem de final de arquivo, EOF;

[6] fecha o arquivo apontado por fp.

**Exemplo 2:** Gravando um arquivo em modo binário.

```
{
FILE *fp; [1]
char *str = "texto de exemplo"; [2]
fp = fopen("c:/temp/fputc.txt", "wb"); [3]
if (fp == NULL) [4]
{
    exit(1);
}
while(*str)
if (!ferror(fp)) fputc(*str++, fp); [5]
fclose(fp); [6]
}
```

Onde:

[1] é uma variável ponteiro do tipo FILE;

[3] abre o arquivo fputc.txt do tipo binário no modo de escrita (w);

[4] testa se o ponteiro é diferente ou igual a nulo, se for NULL interrompe a execução do programa;

[5] enquanto o conteúdo da string apontada por str for diferente de '\0', cada caracter é passado para o arquivo apontado por fp, se não ocorrer uma mensagem de final de arquivo, EOF. Entretanto, para arquivos abertos para operações binárias, um EOF pode ser um

caracter válido. Então é necessário usar a função `ferror` para determinar a ocorrência de um erro;

[6] fecha o arquivo apontado por `fp`.

### 6.6.2. `fgetc()`

Esta função devolve o próximo caracter da stream de entrada na posição atual e incrementa o indicador de posição do arquivo. Se o final do arquivo for alcançado, a função devolverá EOF. Para arquivos binários, testar o final de arquivo com o comando `feof()`. A sintaxe é dada por:

```
fgetc(FILE *pont);
```

Onde, `pont` é o ponteiro para o arquivo.

**Exemplo1:** Lendo um arquivo em modo texto.

```
{  
    FILE *fp; [1]  
    char *ch;  
    ch = (char *)malloc(20 * sizeof(char)); [2]  
    if (!ch) exit(1);  
    fp = fopen("c:/temp/fgetc.txt", "rt"); [3]  
    if (fp == NULL) [4]  
    {  
        exit(1);  
    }  
    while ((*ch++ = fgetc(fp)) != EOF); [5]  
    fclose(fp); [6]  
}
```

Onde:

[1] é uma variável ponteiro do tipo `FILE`;

[2] cria uma área para uma string;

[3] abre o arquivo `fgetc.txt` do tipo texto no modo de leitura (`r`);



[4] testa se o ponteiro é diferente ou igual a nulo, se for NULL interrompe a execução do programa;

[5] enquanto o caracter passado do arquivo apontado por fp, for diferente de EOF, a área de memória apontada por ch recebe os caracteres do arquivo;

[6] fecha o arquivo apontado por fp.

### **6.6.3. Exercício com fputc() e fgetc()**

1- Altere os exemplos acima, para que o usuário possa fornecer através do formulário o nome do arquivo, o local e o texto a ser gravado.

### **6.7. Gravação de strings com fputs()**

Ao invés de se manipular somente caracteres, muitas vezes será necessário gravar toda a string. Para isto usa-se o comando fputs(). Esta função escreve no arquivo o conteúdo da string apontada por str, como apresentado abaixo:

```
fputs(char *str, FILE *pont);
```

Esta função devolve o valor EOF se não conseguir executar a operação.

**Exemplo:** Gravando uma string.

```
{
FILE *fp; [1]

fp = fopen("fputs.txt", "wt"); [3]
if (fp == NULL) [4]
{
exit(1);
}
fputs("texto de teste", fp); [5]
fclose(fp); [6]
}
```

Onde:

[1] é uma variável ponteiro do tipo FILE;

[3] abre o arquivo fputs.txt do tipo texto no modo de escrita (w);

[4] testa se o ponteiro é diferente ou igual a nulo, se for NULL interrompe a execução do programa;

[5] insere a string no arquivo apontado por fp;

[6] fecha o arquivo apontado por fp.

### 6.8. Leitura de strings com fgets()

Ao invés de ler somente caracteres, muitas vezes será necessário ler toda a string. Para isto usa-se o comando fgets(). Esta função é dada por:

```
fgets(string, int num, FILE *pont);
```

Onde a função fgets() lê num-1 caracteres do arquivo apontado por *pont* e coloca-os na string. Os caracteres são lidos até que uma nova linha, ou um EOF seja recebido, ou até que o limite estabelecido em *num* seja atingido. O caracter nulo (\0) é colocado na string após o último valor ser lido do arquivo.

**Exemplo:** Lendo strings.

```
{  
FILE *fp; [1]  
char str[11]; [2]  
fp = fopen("fputs.txt", "rt"); [3]  
if (fp == NULL) exit(1); [4]  
  
fgets(str, 10, fp); [5]  
fclose(fp); [6]  
}
```

Onde:

- [1] é uma variável ponteiro do tipo FILE;
- [2] é a variável que recebe a string lida do arquivo;
- [3] abre o arquivo fputs.txt do tipo texto no modo de leitura (r);
- [4] testa se o ponteiro é diferente ou igual a nulo, se for NULL interrompe a execução do programa;
- [5] lê a string do arquivo apontado por fp;
- [6] fecha o arquivo apontado por fp.

### **6.9. Exercícios com fputs() e fgets()**

- 1- Escreva um programa em C para armazenar em um arquivo, as palavras digitadas por um usuário através de um EditText de um formulário.
- 2- Elabore um programa para que o usuário possa ver as palavras armazenadas pelo programa da letra a.
- 3- Escreva um programa em C para contar a quantidade de palavras de um arquivo texto.
- 4- Escreva um programa para contar a quantidade de linhas de um arquivo texto.
- 5- Escreva um programa para armazenar várias strings fornecidas por um usuário, e que possibilite que o usuário procure por uma string do arquivo. Se a string existir, apresentá-la na tela, caso contrário, apresentar mensagem de erro.

### **6.10. Leitura com fread()**

Usado para ler tipos de dados maiores que um byte. A leitura é feita em bloco, do tamanho especificado pelo programador.

```
{  
FILE *stream; [1]  
char msg[]="isto é um teste"; [2]  
char buf[20]; [3]  
  
fread(buf, strlen(msg)+1,1,stream); [4]  
}
```

Onde:

[1] é uma variável ponteiro do tipo FILE;

[2] é uma variável que contém uma string qualquer;

[3] é uma variável que receberá uma string qualquer lida do arquivo;

[4] comando para ler blocos de dados do disco:

- 1.o parâmetro (buf) é uma variável string qualquer definida pelo programador, que receberá N bytes do arquivo;
- 2.o parâmetro (strlen(msg)+1) é o tamanho do bloco a ser lido;
- 3.o parâmetro (1) é quantidade destes blocos que vão ser lidos ao mesmo tempo.
- 4.o parâmetro é o ponteiro para o arquivo de onde está sendo lido os dados.

### **6.11. Gravação com fwrite()**

Usado para gravar tipos de dados maiores que um byte. A gravação é feita em bloco, do tamanho especificado pelo programador.

```
{
FILE *stream; [1]
char msg[]="isto é um teste"; [2]
...
fwrite(msg, strlen(msg)+1,1,stream); [3]
...
}
```

Onde:

[1] é uma variável ponteiro do tipo FILE;

[2] é uma variável que contém uma string qualquer;

[3] comando para ler blocos de dados do disco:

- 1.o parâmetro (msg) é uma variável string qualquer definida pelo programador, que será gravado no arquivo;
- 2.o parâmetro (strlen(msg)+1) é o tamanho do bloco a ser gravado;
- 3.o parâmetro (1) é quantidade destes blocos que vão ser gravados ao mesmo tempo;
- 4.o parâmetro é o ponteiro para o arquivo de onde estão sendo gravados os dados.

### 6.12. Gravação com fprintf()

```
{  
    FILE *stream; [1]  
    int i = 100;  
    char c = 'C'; [2]  
    float f = 1.234;  
    stream = fopen("c:/temp/teste.$$$", "wt");  
    fprintf(stream, "%d %c %f", i, c, f); [3]  
    fclose(stream);  
}
```

Onde:

[1] é uma variável ponteiro do tipo FILE;

[2] variáveis a serem armazenadas;

[3] comando para gravar dados formatados no disco:

- 1.o parâmetro (stream) é o ponteiro para o arquivo de onde estão sendo gravados os dados;
- 2.o parâmetro é o formato da string a ser gravada;
- 3.o parâmetro (e subsequentes) são as variáveis a serem gravadas.

### 6.13. Leitura com fscanf()

```
{  
    FILE *stream; [1]  
    int i = 0;  
    char c = ' '; [2]  
    float f = 0;  
    stream = fopen("c:/temp/teste.$$$", "rt");  
    fscanf(stream, "%d %c %f", &i, &c, &f); [3]  
    fclose(stream);  
    Edit1->Text = i;  
    Edit2->Text = c;  
    Edit3->Text = f;  
}
```

Onde:

[1] é uma variável ponteiro do tipo FILE;

[2] variáveis que vão armazenar os valores lidos do arquivo;

[3] comando para ler dados formatados no disco:

- 1.o parâmetro (stream) é o ponteiro para o arquivo de onde estão sendo lidos os dados;
- 2.o parâmetro é o formato de como a string foi gravada;
- 3.o parâmetro (e subsequentes) são as variáveis que apontarão para os valores lidos

#### **6.14. Exercícios**

1- Escreva um programa em C que gere e armazene em um arquivo uma matriz de tamanho L x C, com os elementos da diagonal principal preenchidos com o valor '1' e o restante com o valor '0'.

2- Escreva um programa em C que leia o arquivo gerado pelo programa do exercício 7.1 e apresente na tela o conteúdo lido.

3- Escreva um programa em C que leia um arquivo gerado pelo NotePad (bloco de notas do Windows) e grave uma cópia deste arquivo com as linhas em ordem invertida. (a última linha do arquivo é gravada por primeiro, e assim por diante...)

4- DESAFIO: Escreva um programa em C que leia um arquivo de texto criado pelo NotePad, e converta este arquivo para o formato HTML (padrão de arquivo da internet), de forma que possa ser aberto em um browser.

## 7. REGISTROS

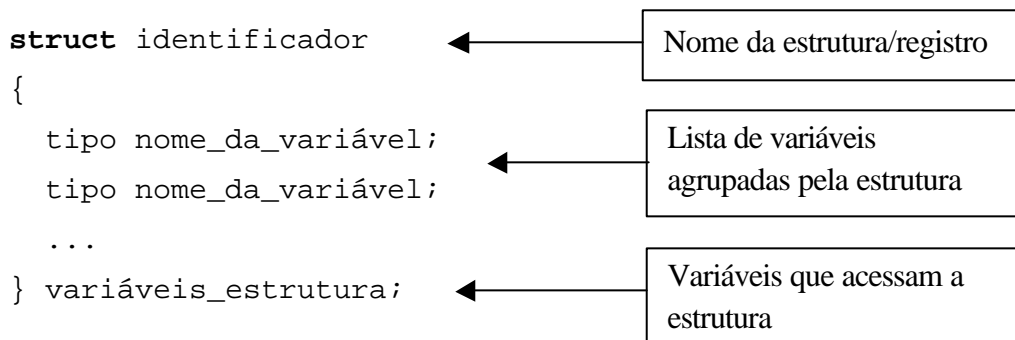
---

### 7.1. Definição

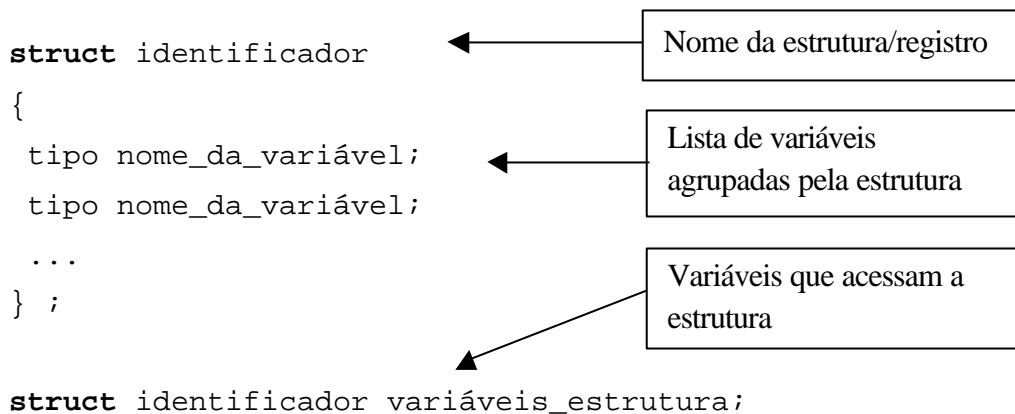
Um registro é uma coleção de uma ou mais variáveis, possivelmente de tipos diferentes, colocadas juntas sob um único nome. Os registros também são chamados de estruturas. As variáveis que compreendem a estrutura são chamadas de membros da estrutura (ou elementos, ou campos).

### 7.2. Inicialização

Um registro é declarado da seguinte forma:



Também pode ser declarado como:



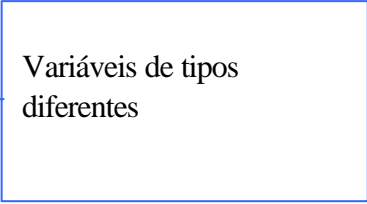
### 7.2.1. Exemplo 1

```

struct addr
{
  char nome[30];
  char rua[40];
  char cidade[20];
  char estado[3];
  unsigned int cep;
};

struct addr endereco; // cria uma variável endereço do tipo addr.
...

```



### 7.2.2. Exemplo 2

```

struct addr
{
  char nome[30];
  char rua[40];
  char cidade[20];
  char estado[3];
  unsigned long int cep;
} endereco, informacao ; // define variáveis do tipo addr

```

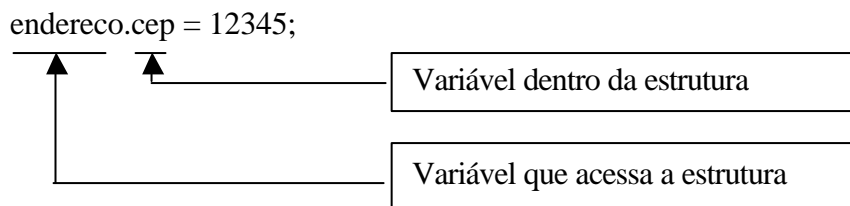
### 7.3. Acesso aos elementos da estrutura

Cada elemento é referenciado por um operador (o ponto "." é normalmente usado para isso).

#### **Exemplo:**

Para atribuir o valor 12345 ao elemento "cep" da estrutura, basta referenciá-lo da seguinte forma:





E, de forma geral:

```
nome_da_estrutura.nome_do_campo = valor;
```

### 7.4. Exercício

Escreva um programa em C que leia os dados de cadastro de uma pessoa através de um formulário, e armazene as informações em um registro.

### 7.5. Matrizes de estruturas

É possível criar uma matriz de estruturas da seguinte forma:

```
struct addr endereco[100];
```

Esta declaração cria um vetor com 100 endereços, contendo para cada um, os elementos da estrutura.

#### 7.5.1.Exemplo

<pre>struct dados {   double notas[4];   char *nome; } aluno;</pre>	<p>Este exemplo define uma estrutura (registro) que cria um <u>aluno</u> do tipo <u>struct dados</u>.</p> <p><b>aluno.nome</b> = pode receber um nome para aluno. exemplo: aluno.nome = "Rudek";</p> <p><b>aluno.notas[x]</b> = armazena uma nota na posição x do vetor.</p>
---	--

<pre>struct dados { double notas[4]; char *nome; } aluno[10];</pre>	<p>De forma mais dinâmica, podemos criar registros para N alunos. Assim,</p> <pre>aluno[1].nome = "Pedro"; aluno[1].notas[0] = 10;  aluno[2].nome = "João" aluno[2].nota = 6;  etc...</pre>
---	---

### 7.5.2. Exercício

1 - Elabore um programa em C para calcular a média das 4 notas bimestrais de cada aluno (usando registros), para um professor que tenha 3 turmas de 5 alunos.

The image shows a screenshot of a Windows application window titled "Form1". The window has a standard Windows XP-style title bar with minimize, maximize, and close buttons. The main area of the form is light gray and contains the following elements:

- A label "Turma" followed by a text box containing the number "0".
- A label "Nome" followed by a text box containing the text "Fulano de Tal".
- A label "Notas" followed by four separate text boxes containing the values "7.5", "8.0", "6.0", and "8.5" respectively.
- A single "OK" button centered at the bottom of the form.

**Algoritmo do registro**

```

estrutura dados
{
real notas[4];
caracter *nome;
} aluno;

estrutura classes
{
estrutura dados aluno[5];
} turma[3];

```

2 - Escreva um programa em C que armazene 10 registros com as informações do formulário abaixo. Use botões para "navegar" entre os registros.

3- Inclua no programa do exercício 2 a possibilidade de procurar um registro (dentre os digitados) pelo nome da pessoa, e apresentar seus dados na tela.

4- Inclua no programa do exercício 2 a possibilidade de excluir um registro que possua o campo nome igual ao valor passado pelo usuário.

## 7.6. Uso de TYPEDEF

A linguagem C permite que o programador defina novos nomes aos tipos de dados já existentes. A forma geral de uso de typedef é a seguinte:

**typedef** tipo novonome;

### 7.6.1. Exemplo

No Programa:

```
typedef float TIPOREAL;
```

No Botão:

```
{  
  TIPOREAL x = 0;  
  char M[20];  
  sprintf(M,"valor de x é %f",x);  
}
```

### 7.6.2. Exemplo 2

No Programa:

```
typedef struct  
{  
  char Nome[80];  
  int Numero;  
} ALUNO;
```

No Botão:

```
{
ALUNO turma[20];
char M[20];
printf("O nome do primeiro aluno é %s", turma[0].Nome );
}
```

## 7.7. Gravação e Leitura de Registros

Qual é o tamanho de um registro? Normalmente um valor maior que um byte. Ou seja, tem-se um bloco de dados de tamanho dependente da quantidade de campos e de seus tipos de dados. Sendo assim, a melhor forma de armazenar/ler registros em arquivos é através dos comandos fwrite() e fread().

### 7.7.1 Exemplo

```
typedef struct
{
int idade;
char *nome;
} Dados;

Dados regDados;
FILE *fp;
...
fwrite(&regDados, sizeof(Dados),1, fp);
...
fread(&regDados, sizeof(Dados),1, fp);
```

### 7.7.2.Exercício

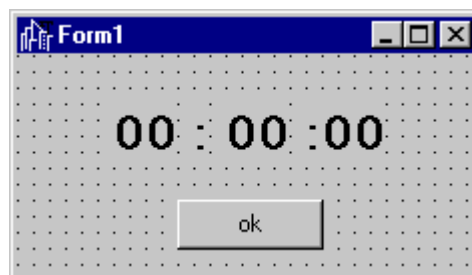
1 - Escreva um programa em C que armazene em um arquivo os dados cadastrais de n pessoas, e possibilite recuperar o registro do arquivo através da busca pelo nome da pessoa.

## 7.8. Ponteiros para registros

Muitas vezes será necessário atualizar o conteúdo de uma estrutura de qualquer parte do programa.

### 7.8.1.Exemplo

O exemplo a seguir, apresenta um programa que simula um cronômetro:



Onde o programa é dado por:

```
#define DELAY 128000

typedef struct
{
  int h;
  int m;
  int s;
} relógio;

void mostra(relógio *);
void atualiza(relógio *);
void delay(void);

//-----
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
{
}
```

```
//-----  
void delay(void)  
{  
    unsigned long int y=1;  
    for (y=1; y<DELAY; ++y);  
}  
  
//-----  
void atualiza(relogio *r)  
{  
    r->s++;  
    if (r->s == 60) // testa segundos  
    {  
        r->s = 0;  
        r->m++;  
        if (r->m == 2) exit(0);  
    }  
    if (r->m == 60) // testa minutos  
    {  
        r->m = 0;  
        r->h++;  
    }  
    if (r->h==24) r->h = 0; // testa horas  
}  
  
//-----  
void mostra(relogio *r)  
{  
    char hora[3];  
    char min[3];  
    char seg[3];  
  
    itoa(r->h,hora,10);  
    itoa(r->m,min,10);  
    itoa(r->s,seg,10);  
  
    Form1->Label1->Caption = hora;  
    Form1->Label2->Caption = min;  
    Form1->Label3->Caption = seg;  
}
```

```
//-----  
void __fastcall TForm1::Button1Click(TObject *Sender)  
{  
    relógio hor_atual;  
  
    hor_atual.h = 0;  
    hor_atual.m = 0;  
    hor_atual.s = 0;  
  
    for (;;)   
    {  
        atualiza(&hor_atual);  
  
        mostra(&hor_atual);  
  
        for (int z=1;z < 400; ++z)  
        {  
            delay(); // atrasa execução do programa  
        }  
  
        Refresh();  
    }  
}  
//-----  
void __fastcall TForm1::FormClick(TObject *Sender)  
{  
    Close();  
}  
//-----
```



## 8. GRÁFICOS EM C

---

### 8.1. INTRODUÇÃO

As operações com gráficos em C, estão relacionados com o objeto Canvas.

Este objeto contém as propriedades de desenho, como:

- **Font:** Especifica o tipo da fonte, quando se aplica texto sobre uma imagem. As propriedades que podem ser usadas se referem à face, a cor, ao tamanho, e ao estilo;
- **Brush:** Determina a cor e o padrão usado por Canvas para preencher formas e o plano de fundo;
- **Pen:** Especifica o tipo da caneta que o Canvas usa para desenhar linhas e formas;
- **PenPos:** Especifica a posição corrente de uma caneta;
- **Pixels:** Especifica a cor da área de pixels dentro do ClipRect corrente.

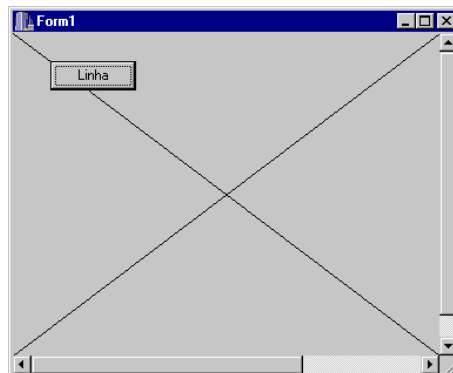
Estas propriedades estão apresentadas em mais detalhes na descrição do objeto Canvas, no help do C Builder.

### 8.2. Desenhando Linhas

O comando `LineTo` desenha uma linha a partir da posição atual `PenPos` até o ponto especificado por `X` e `Y`, e altera a posição da caneta para `(X, Y)`. O comando `MoveTo` altera a posição corrente do ponto `(X,Y)`.

```
void __fastcall TForm1::LinhaClick(TObject *Sender)
{
    Canvas->MoveTo(0,0);
    Canvas->LineTo(ClientWidth, ClientHeight);
    Canvas->MoveTo(0, ClientHeight);
    Canvas->LineTo(ClientWidth, 0);
}
```

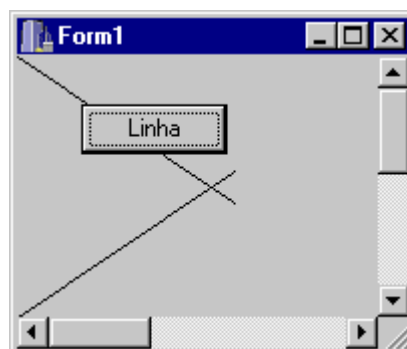
Ao executar os comandos do botão linha, obtém-se no formulário o desenho das linhas.



Observe que a instrução ClientWidth, usada no exemplo acima, permite que as linhas sejam desenhadas dentro dos limites de tamanho do formulário.



Observa-se também o seguinte problema: ao se redimensionar a janela, (por exemplo, diminuir a largura e em seguida aumentá-la) nota-se que uma parte do desenho das linhas é apagado:



Isto ocorre porque o C++, no Windows, associa a reconstrução dos objetos de tela usando o evento OnPaint.

Assim, se alterarmos o programa para:

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
    Canvas->MoveTo(0,0);
    Canvas->LineTo(ClientWidth, ClientHeight);
    Canvas->MoveTo(0, ClientHeight);
    Canvas->LineTo(ClientWidth, 0);
}
```

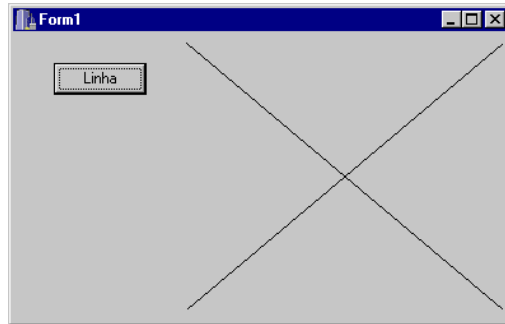
E, no botão linha, apenas o comando que força a chamado de OnPaint:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Refresh();
}
```

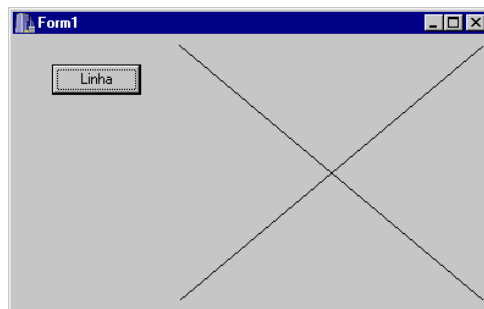
### **8.3. Usando o PaintBox**

É possível especificar uma área para desenho, dentro do formulário para que os elementos gráficos não sobrescrevam os componentes do formulário. Então, associado ao evento Paint do formulário, temos:

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
    PaintBox1->Canvas->MoveTo(0,0);
    PaintBox1->Canvas->LineTo(PaintBox1->ClientWidth, PaintBox1->ClientHeight);
    PaintBox1->Canvas->MoveTo(0,PaintBox1->ClientHeight);
    PaintBox1->Canvas->LineTo(PaintBox1->ClientWidth, 0);
}
```

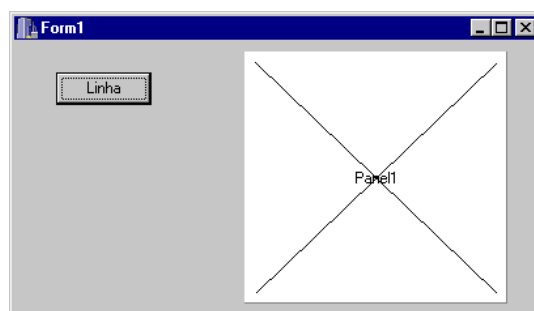


```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
    PaintBox1->Canvas->MoveTo(0,0);
    PaintBox1->Canvas->LineTo(PaintBox1->ClientWidth, PaintBox1-
ClientHeight);
    PaintBox1->Canvas->MoveTo(0,PaintBox1->ClientHeight);
    PaintBox1->Canvas->LineTo(PaintBox1->ClientWidth, 0);
}
```



#### **8.4. Componente Panel**

O objeto Panel permite definir um grupo de objetos de formulário associados a ele. Neste caso usamos para criar uma área visível para o desenho.



**Obs.:** Muitas vezes, é necessário usar o evento Paint do componente, ao invés de usar o Paint do formulário. Então poderíamos escrever assim:

```
void __fastcall TForm1::PaintBox1Paint(TObject *Sender)
{
    PaintBox1->Canvas->MoveTo(0,0);
    PaintBox1->Canvas->LineTo(PaintBox1->ClientWidth, PaintBox1->ClientHeight);
    PaintBox1->Canvas->MoveTo(0,PaintBox1->ClientHeight);
    PaintBox1->Canvas->LineTo(PaintBox1->ClientWidth, 0);
}
```

### 8.5. Desenhando Retângulos

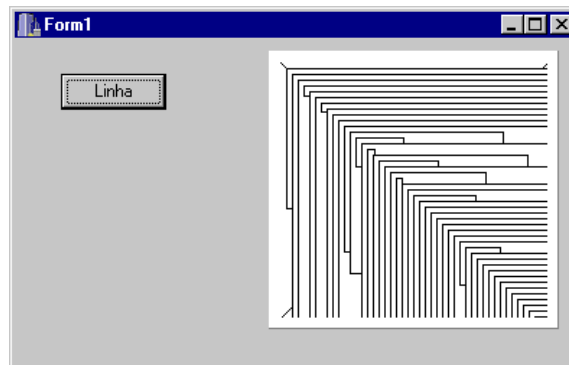
Para ilustrar o desenho de retângulos, usaremos um exemplo do help do C++ Builder. Onde criamos duas variáveis globais x e y, e inicializamos o formulário da seguinte maneira:

```
void __fastcall TForm1::FormActivate(TObject *Sender)
{
    WindowState = wsMaximized;
    Canvas->Pen->Width = 5;
    Canvas->Pen->Style = psDot;
    Timer1->Interval = 50;
    randomize();
}
```

E, associado a um timer:

```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
    x+=4;
    y+=4;
    Canvas->Pen->Color = random(65535);
    Canvas->Rectangle(x, y, x + random(400), y + random(400));
    if(x > 100)
        Timer1->Enabled = false;
}
```

Exemplo do resultado dos comandos acima:

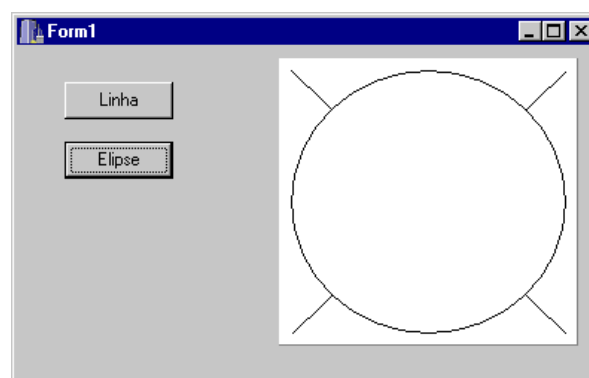


## 8.6. Desenhando Elipses

Para ilustrar o desenho de elipses, usaremos um exemplo do help do C++ Builder:

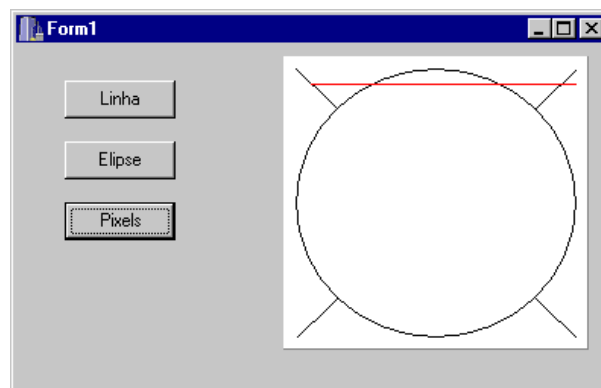
Ao clicar no botão "Elipse":

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    PaintBox1->Canvas->Ellipse(0, 0, PaintBox1->Width, PaintBox1->Height);
}
```



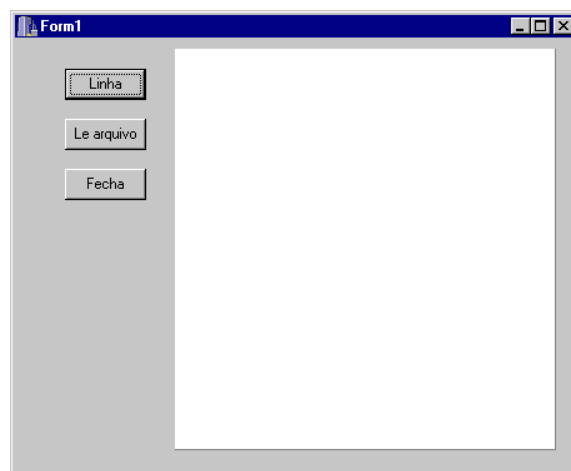
### 8.7. Desenhando Pontos (Pixels)

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    for(int i = 10; i < 200; i++)
        Canvas->Pixels[i][10] = clRed;
}
```



### 8.8. Exemplo

Dado o formulário abaixo, desenvolva um programa em C que desenhe linhas de acordo com coordenadas especificadas pelo usuário. As coordenadas devem ser salvas em um arquivo, para que possam ser recuperadas posteriormente.



## Solução

a) Antes de qualquer construção gráfica é necessário determinar as variáveis que irão armazenar os pontos (coordenadas de tela). Para construir uma linha, são necessário 2 pontos, representados por 4 valores numéricos (x1,y1) e (x2,y2) dados como pontos iniciais e finais da linha, respectivamente. Para N linhas desenhadas será necessário um espaço de (Nx4 valores). A quantidade de pontos é limitada pelo tamanho de memória alocado para armazená-los. Então se definirmos um registro para armazenar os pontos, teremos por exemplo:

```
typedef struct
{
    int X1;
    int X2;
    int Y1;
    int Y2;
} pontos;
```

A estrutura "pontos" definida acima pode armazenar apenas um ponto inicial e apenas um ponto final, o que representa apenas um segmento de reta. Para que seja possível armazenar mais do que uma reta, precisa-se várias estruturas como esta. Para obter "mais" estruturas iguais ao registro "ponto", podemos definir um vetor, ou alocar um espaço dinamicamente através de um ponteiro para esta estrutura, como mostra o exemplo abaixo:

```
typedef struct
{
    int X1; // coordenadas do ponto inicial da reta
    int X2;
    int Y1; // coordenadas do ponto final da reta
    int Y2;
} pontos;

pontos *coord; // ponteiro para o registro
int pos = 0; // posição do registro
int cont = 0; // quantidade de registros
```



E quando o programa começa a ser executado, cria-se a área de memória para armazenar os pontos. No trecho de programa abaixo, indicado pela linha [1], o ponteiro "coord" aponta para região de memória alocada pelo comando malloc(), cujo tamanho é de 100 vezes o tamanho do registro "pontos".

```

__fastcall TForm1::TForm1(TComponent* Owner): TForm(Owner)
{
    coord = (pontos *) malloc(100*(sizeof(pontos))); //[1]
    if (coord == NULL)
    {
        Application->MessageBox("Não há Memória", "Erro", MB_OK);
        exit(1); // termina o programa
    }
}

```

Para gerar uma linha é necessário associar os comandos LineTo e MoveTo a um botão que execute a ação. O formulário abaixo recebe os parâmetros para traçar as linhas,

E, em seguida são passados para o Form1:

```

void __fastcall TForm1::LinhaClick(TObject *Sender)
{
    Form2->ShowModal(); // abre 2.o formulário para entrada de dados
    coord[pos].X1 = atoi(Form2->eX1->Text.c_str());
    coord[pos].X2 = atoi(Form2->eX2->Text.c_str());
    coord[pos].Y1 = atoi(Form2->eY1->Text.c_str());
    coord[pos].Y2 = atoi(Form2->eY2->Text.c_str());
}

```

```

FILE *arq;
if(( arq = fopen("C:\\TEMP\\QUAD.TXT","at"))== NULL) exit(1);
else
{
    fwrite(&coord[pos],sizeof(pontos),1,arq);
    fclose(arq);
}
pos++;
}

```

Para recuperar os dados, deve-se ler as informações do arquivo, como abaixo:

```

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    FILE *arq;
    cont = 0;
    if(( arq = fopen("C:\\TEMP\\quad.TXT","rt"))== NULL) exit(1);
    else
    {
        do
        {
            fread(&coord[cont],sizeof(pontos),1,arq);
            cont++;
        }
        while(!feof(arq));
        fclose(arq);
    }
    pos = cont;
    PaintBox1->Refresh();
}

```

Efetivamente a função que desenha na tela deve ser executada na chamada de um evento `OnPaint`.

```
void __fastcall TForm1::PaintBox1Paint(TObject *Sender)
{
    TPaintBox *pCor = (TPaintBox *)Sender;
    int x;
    for(x=0; x <= pos;x++)
    {
        pCor->Canvas->Pen->Color = (Graphics::TColor) random(65535);
        PaintBox1->Canvas->MoveTo(coord[x].X1,coord[x].Y1);
        PaintBox1->Canvas->LineTo(coord[x].X2,coord[x].Y2);
    }
}
```

### **8.9 Exercícios**

1- Modifique o programa acima para apresentar no formulário a posição do mouse quando movimentado dentro do PaintBox.

Resp: Testar o evento MouseMove do PaintBox

2- Modifique o exemplo acima, adicionando a possibilidade de desenhar retângulos e elipse (1 botão para retângulo e 1 botão para Elipse).

Resp: Use os comandos Rectangle() e Ellipse()

3- Modifique o exemplo acima, adicionando a possibilidade de inserir pontos de acordo com coordenadas definidas pelos usuários. Exemplo:

PaintBox1->Canvas->Pixels[X1][Y1] = clRed;

## 9. LISTAS LINEARES

---

### 9.1. Fila

#### 9.1.1. Definição

Uma FILA é uma lista linear de dados acessada na ordem em que o primeiro elemento que entra na lista é o primeiro a ser retirado. Também é chamada de lista FIFO (First In, First Out). Um exemplo comum, é uma fila de banco. Elementos podem ser adicionados às filas e retirados. A cada adição, conseqüentemente, aumenta a fila, e a cada saída, diminui-se a fila; então uma fila pode ficar com comprimento 0.

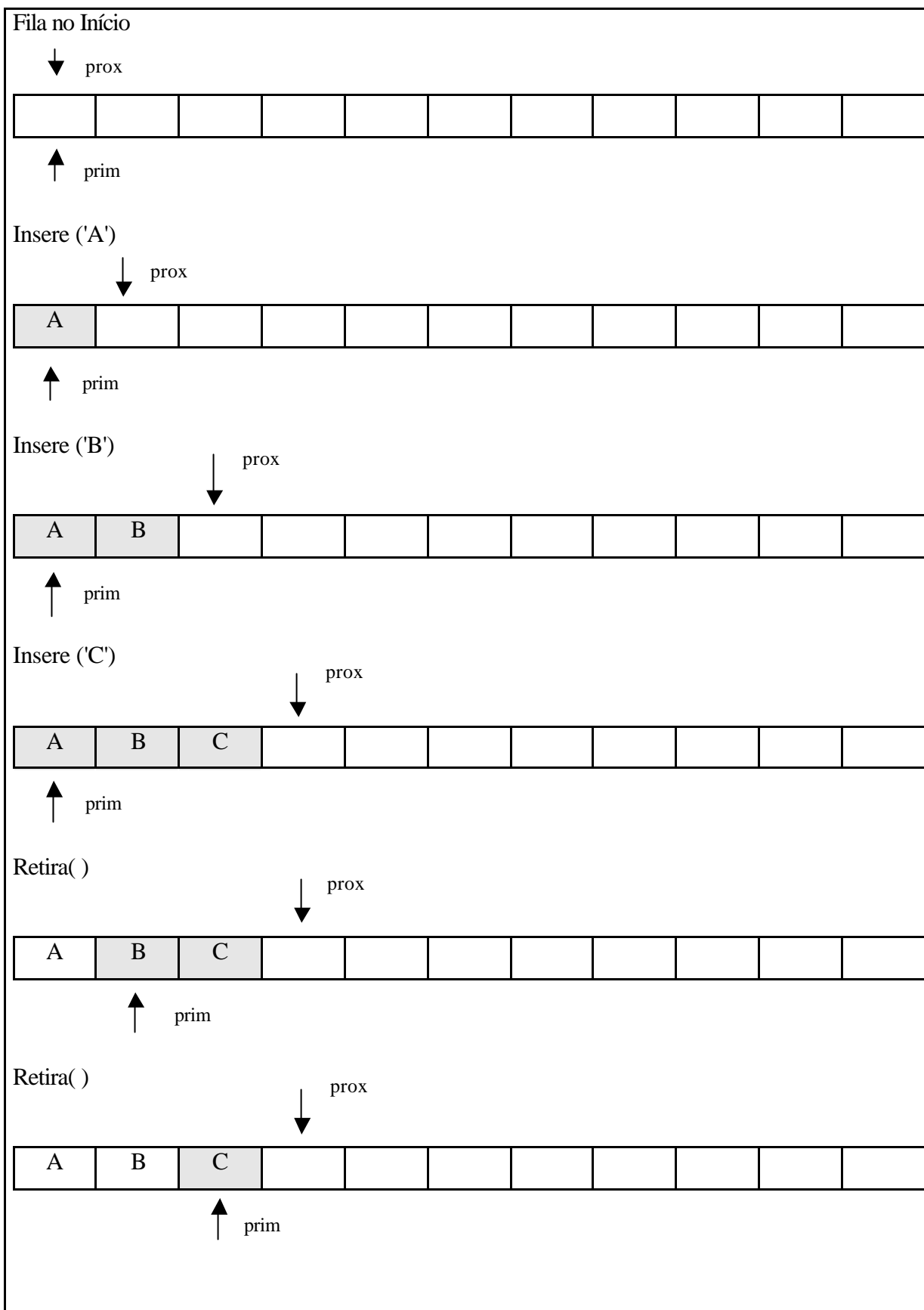
#### 9.1.2. Objetivo

As filas são muito usadas em programação. Servem principalmente para simulações e distribuição de eventos como diagramas PERT, e bufferização de Entrada e Saída (Fila de impressão).

#### 9.1.3. Exemplo

Supondo duas funções **Inserer()** e **Retira()**, que inserem e retiram respectivamente elementos da fila, temos:

Ação	Conteúdo da Fila
Inserer(A)	[A]
Inserer(B)	[A B]
Inserer(C)	[A B C]
Retira( )	[B C]
Inserer(D)	[B C D]
Retira( )	[C D]
Retira( )	[D]
Retira( )	[]



Exemplo de função em C que executa os comandos de inserção apresentadas no item 9.1.3, acima:

```
//-----
#define MAX 20
char *p[MAX]; // vetor de ponteiros
int prox=0; // próximo elemento livre da fila
int prim=0; // primeiro elemento da fila
// insere elementos na fila
void Insere(char *q)
{
    if (prox==MAX)
    {
        Application->MessageBox("Lista Cheia","Mensagem",MB_OK);
        return; //lista cheia
    }

    // se Insere recebe 'A'
    p[prox] = q; // p[0] aponta para o endereço de 'A' "&q"
    prox++; // próxima posição vazia da lista.
}

```

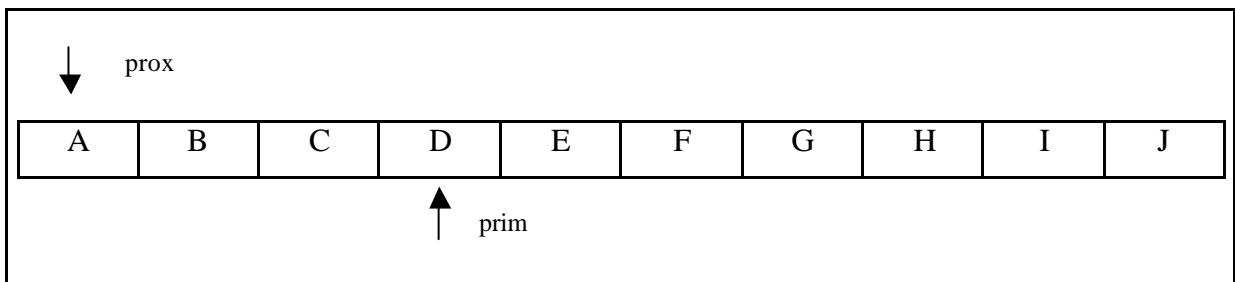
Exemplo de função em C que executa os comandos de remoção apresentadas no item 9.1.3, acima:

```
// retira elementos da fila, esta função retorna um ponteiro
char *retira()
{
    if (prim==prox)
    {
        Application->MessageBox("Lista Vazia","Mensagem",MB_OK);
        return NULL;
    }
    prim++; // início da fila para o segundo elemento
    return p[prim-1]; // retorna o elemento retirado da fila
}

```

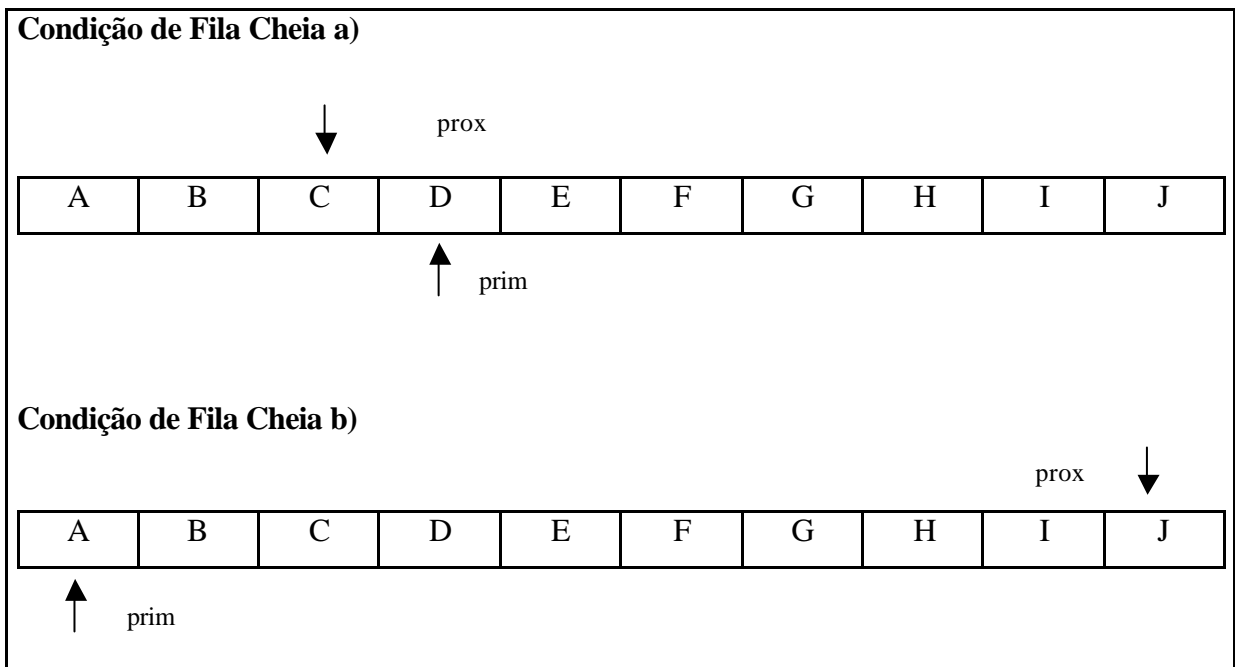
## 9.2. Fila Circular

Quando o limite do tamanho da fila é atingido, é possível retornar ambos os índices (**prim** e **prox**) para o início do vetor utilizado. Assim pode-se ter um aproveitamento maior da lista.



Baseado no exemplo anterior, a fila estará cheia se:

- a) **prox** está uma posição anterior a **prim**, ou se;
- b) **prox** estiver no final e **prim** no início da lista.



Um exemplo de função, que implementa inserção de dados em uma lista circular é dado abaixo:

```
//-----
#define MAX 20
char *p[MAX]; // vetor de ponteiros
int prox=0; // proximo elemento livre da fila
int prim=0; // primeiro elemento da fila

// insere elementos na fila
void Insere(char *q)
{
    if ((prox+1==prim)|| (prox+1==MAX && !prim))
    {
        Application->MessageBox("Lista Cheia", "Mensagem", MB_OK);
        return; // lista cheia
    }

    // se Insere recebe 'A'
    p[prox] = q; // p[0] aponta para o endereco de 'A' "&q"
    prox++; // proximo posicao vazia da lista.
    if (prox==MAX) prox=0; // reinicia
}

```

Um exemplo de função, que implementa remoção de dados em uma lista circular é dado abaixo:

```
// retira elementos da fila, esta função retorna um ponteiro
char *retira()
{
    if (prim==MAX) prim=0; // reinicia
    if (prim==prox)
    {
        Application->MessageBox("Lista Vazia", "Mensagem", MB_OK);
        return NULL;
    }
    prim++; // início da fila para o segundo elemento
    return p[prim-1]; // retorna o elemento retirado da fila
}

```



### 9.3. Pilha

#### 9.3.1. Definição

Uma PILHA é o inverso de uma FILA. Baseia-se no princípio LIFO (Last IN, First OUT ), ou seja o último elemento adicionado é o primeiro a ser retirado. Uma pilha de pratos é um exemplo. Pilhas são muito usadas em compiladores. As duas operações para armazenar e retirar elementos da pilha são tradicionalmente denominadas push e pop.

Ação	Conteúdo da Fila
Push(A)	[ A ]
Push(B)	[ B A ]
Push(C)	[ C B A ]
Pop()	[ B A ]
Push(D)	[ D B A ]
Pop()	[ B A ]
Pop()	[ A ]
Pop()	[ ]

#### 9.3.2. Exemplo

```
//-----
#define MAX 20
int *p;
int *topo;
int *tama;

void push(int i)
{
    if(p > tama)
    {
        Application->MessageBox("Pilha Cheia","Mensagem",MB_OK);
        return; //pilha cheia
    }
}
```

```

*p = i; // falta alocar memoria para p
p++; // proxima posicao da pilha
}

int pop(void)
{
p--;
if(p < topo)
{
Application->MessageBox("Pilha Vazia", "Mensagem", MB_OK);
return 0; //pilha vazia
}
return *p;
}

/** Antes que estas funções possam ser usadas, uma região de memória livre
deve ser alocada com MALLOC(). O endereço de início dessa região deve ser
atribuída a topo, e o endereço final a tama.* */

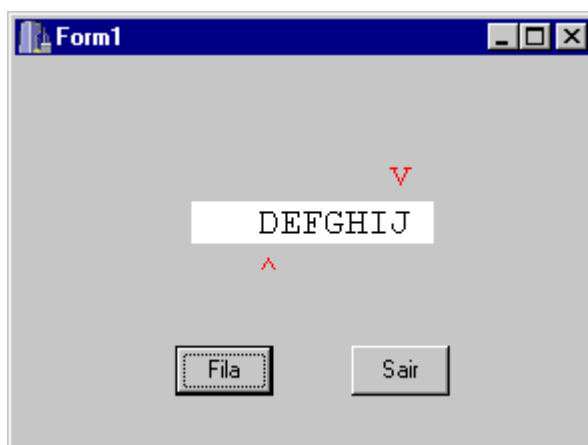
void main(void)
{
// obtém memória para a pilha
p = (int *) malloc(MAX * sizeof(int));
if (!p)
{
Application->MessageBox("Falha de Alocação", "", MB_OK);
exit(1);
}
topo = p; // topo da pilha
tama = p + MAX -1; // tamanho máximo da pilha
}

```

#### 9.4. Exercícios

1- Desenvolva um programa em C para simular as operações de inserção e retirada de elementos de uma fila. A partir do formulário abaixo, a cada clique do botão "Fila", um

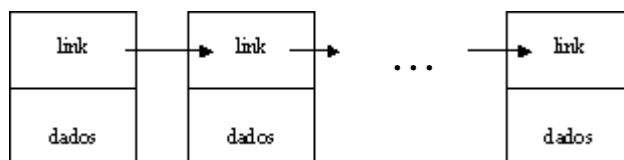
elemento (letras do alfabeto) é inserido ou retirado aleatoriamente. As posições dos ponteiros também são apresentadas e indicam a posição inicial e final da fila.



- 2- Altere o exercício 4.1 acima para que os elementos sejam inseridos ou retirados da fila, associados a um tempo definido por um timer.
- 3- Altere o exercício 4.1 para que a fila apresentada seja um fila circular.
- 4- Desenvolva um programa para simular a inserção e retirada de elementos de uma pilha.

### 9.5. Listas Encadeadas

Em uma lista encadeada pode-se acessar elementos da lista de forma aleatória, ou seja, inserir e remover elementos em qualquer parte da lista. Cada elemento é um nó que contém informação sobre o elo de ligação entre cada um da lista. A característica mais importante deste tipo de estrutura é sua forma dinâmica. Os nós são inseridos ou removidos de acordo com a necessidade, em tempo de execução.



Os nós conectados através de ponteiros formam uma lista encadeada. Cada nó é composto basicamente por duas informações: um campo de dados e um campo de ponteiros, onde se armazenam os endereços das conexões entre os nós.

**9.6. Exemplo**

Criar uma lista encadeada com os seguintes elementos:

Elemento A	Elemento B	Elemento C
Nome: Fulano1 Idade: 30	Nome: Fulano2 Idade: 31	Nome: Fulano3 Idade: 32

```
//Declarado como global
typedef struct aluno
{
    char nome[50];
    int idade;
    struct aluno *prox;
}

aluno *inicio, *pt_aux;
// associado a um botão:
{
    pt_aux = (aluno *) malloc(sizeof(aluno));
    strcpy(pt_aux->nome, "Fulano1");
    pt_aux->idade = 30;
    pt_aux->prox = NULL;
    inicio = pt_aux;
    pt_aux = (aluno *) malloc(sizeof(aluno));
    strcpy(pt_aux->nome, "Fulano2");
    pt_aux->idade = 31;
    pt_aux->prox = inicio;
    inicio = pt_aux;
    pt_aux = (aluno *) malloc(sizeof(aluno));
    strcpy(pt_aux->nome, "Fulano3");
    pt_aux->idade = 32;
    pt_aux->prox = inicio;
    inicio = pt_aux;
}
```

### 9.7. Exercício

1- Altere o exemplo acima para uma função que execute a inserção de elementos em uma lista encadeada.

### 9.8. Exemplo

Duas coordenadas de tela X e Y, podem estar definidas numa estrutura. Se quisermos guardar uma sequência de pontos, podemos criar uma lista encadeada para armazená-los, assim que forem gerados. Uma estrutura para isto poderia ser da seguinte forma:

```
struct pontos
{
    int X;
    int Y;
    struct pontos *prox;
};
```

Os ponteiros de uma lista encadeada apontam nós que são semelhantes em termos de tipos de dados. Assim, os ponteiros são declarados como sendo elementos que apontam para o tipo da estrutura que define o nó.

Neste exemplo temos o ponteiro \*prox, que aponta elementos do tipo struct pontos.

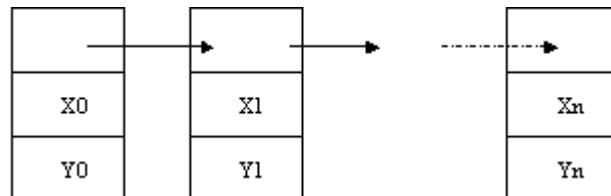
Também é necessário conhecer o início da lista. Pode-se definir um ponteiro para isto também, como:

```
struct pontos *ini; // é um ponteiro para estrutura pontos
```

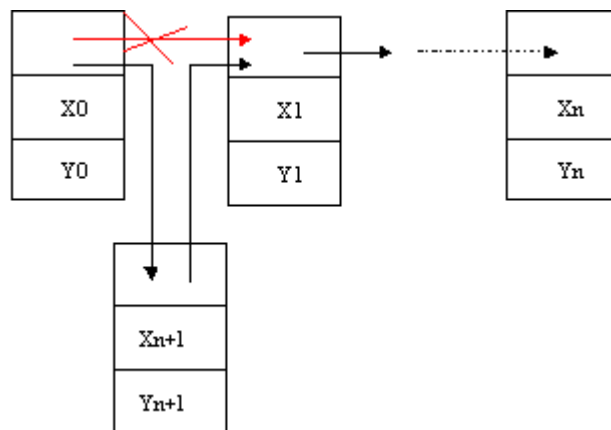
### 9.9. Operações com Lista Encadeada

Dada a lista abaixo, pode-se efetuar operações de inserção e remoção de nós desta lista como ilustrado:

Lista Inicial:

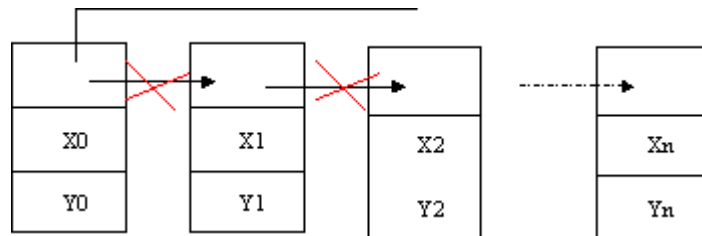


Operação de Inserção:



A codificação em C para uma operação de inserção pode ser:

```
int insere(struct pontos *elemento, struct pontos *pos)
{
    if (elemento != NULL)
    {
        elemento->prox = (pos != NULL) ? pos->prox : pos;
        if (pos != NULL) pos->prox = elemento;
        return 1; // inserção OK
    }
    else return 0; // falha na inserção
}
```

Operação de Remoção:

A codificação em C para uma operação de remoção pode ser:

```
int remove(struct pontos *elemento)
{
    struct pontos *ant;
    if (elemento == NULL) return 0;
    for(ant = ini; ant != NULL; ant = ant->prox)
    {
        if (ant->prox == elemento)
        {
            ant->prox = elemento->prox;
            free((struct pontos *));
            break;
        }
    }
    return 1;
}
```

9.10. Exemplo

```
#include <stdio.h>
#define ALOCA (struct aluno *) malloc(sizeof(struct aluno))
struct aluno
{
    char nome[20];
    int idade;
    struct aluno *prox;
};
```

```

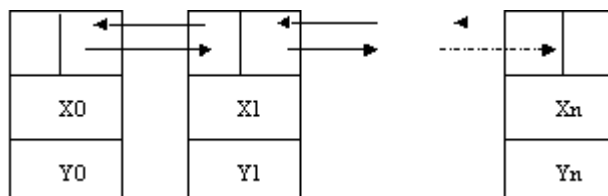
//-----
struct aluno *nova(void)
{
    struct aluno *x;
    x = ALOCA;
    x->prox = 0;
    return (x);
}
//-----
struct aluno *insere(struct aluno *pt)
{
    if (pt->prox == NULL)
    {
        strcpy(pt->nome, Form1->Edit1->Text.c_str());
        pt->idade = atoi(Form1->Edit2->Text.c_str());
        pt->prox = nova();
    }
    else insere(pt->prox);
    return (pt);
}
//-----
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    inicio = ALOCA;
    proximo = ALOCA;
}
//-----
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    proximo = insere(proximo);
}

```



### 9.11. Listas Duplamente Encadeadas

Cada elemento da lista tem uma ligação com o elemento seguinte e a referência do elemento anterior da lista. Ou seja, cada elemento aponta para o elemento seguinte e também para o anterior. Assim pode-se percorrer a lista em ambas as direções.

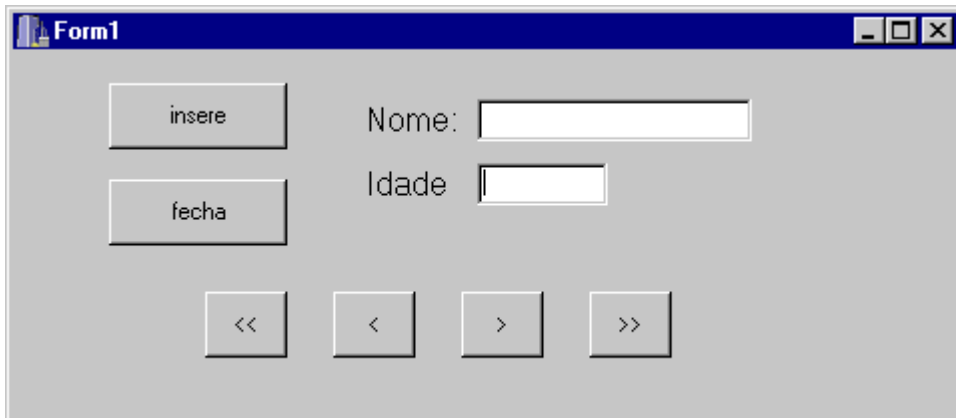


### 9.12. Exemplo

```
typedef struct lista_dupla
{
    int X;
    int Y;
    lista_dupla *ini;
    lista_dupla *fim;
};
lista_dupla lista_dp;
```

Para controlar o acesso a lista ligada pode-se usar uma célula especial que contém informações do comprimento da lista, o endereço da primeira célula da lista e o endereço da última célula da lista.

```
struct info_lista
{
    int comprimento;
    lista_dp *inicio;
    lista_dp *fim;
};
typedef struct info_lista;
```

**9.13. Exemplo**

```

#define ALOCA (struct aluno *) malloc(sizeof(struct aluno))

struct aluno
{
    char nome[20];
    int idade;
    struct aluno *prox;
    struct aluno *ant;
};
struct aluno *registro;
struct aluno *inicio;
struct aluno *ultimo;
struct aluno *pos;
//-----
struct aluno *nova(void)
{
    struct aluno *x;
    x = ALOCA;
    x->ant = registro;
    x->prox = 0;
    ultimo = x;
    return (x);
}
//-----
struct aluno* insere(struct aluno *pt)
{
    if (pt->prox == NULL)

```

```

{
    strcpy(pt->nome, Form1->Edit1->Text.c_str());
    pt->idade = atoi(Form1->Edit2->Text.c_str());
    pt->prox = nova();
}
else
{
    insere(pt->prox);
}
return (pt->prox);
}
//-----
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
{
    inicio = registro = ALOCA;
}
//-----
--
void __fastcall TForm1::bInsereClick(TObject *Sender)
{
    registro = insere(registro);
    Edit1->Text = "";
    Edit2->Text = "";
}
//-----
void __fastcall TForm1::bApagaClick(TObject *Sender)
{
    free(registro);
    Close();
}
//-----
void __fastcall TForm1::bAnteriorClick(TObject *Sender)
{
    pos = (registro->ant != NULL)? registro->ant:ultimo;
    Edit1->Text = pos->nome;
    Edit2->Text = pos->idade;
    registro = pos;
}

```

```
//-----  
void __fastcall TForm1::bInicioClick(TObject *Sender)  
{  
    pos = inicio;  
    Edit1->Text = pos->nome;  
    Edit2->Text = pos->idade;  
    registro = pos;  
}  
//-----  
void __fastcall TForm1::bUltimoClick(TObject *Sender)  
{  
    pos = ultimo;  
    Edit1->Text = pos->nome;  
    Edit2->Text = pos->idade;  
    registro = pos;  
}  
//-----  
void __fastcall TForm1::bProximoClick(TObject *Sender)  
{  
    pos = (registro->prox != NULL)? registro->prox:inicio;  
    Edit1->Text = pos->nome;  
    Edit2->Text = pos->idade;  
    registro = pos;  
}  
//-----
```

## 10. RECURSIVIDADE

---

### 10.1. INTRODUÇÃO

Na linguagem C, funções podem chamar a si mesmas. A função é dita recursiva se um comando no corpo da função a chama novamente. A cada chamada, obtém-se um novo conjunto de todas as variáveis usadas na função, independente do conjunto anterior.

### 10.2. EXEMPLO

Pode-se por exemplo fazer um programa para calcular o fatorial de um número de duas formas. O exemplo 1 apresenta uma solução sem o uso de recursividade e o exemplo 2 usa o conceito de recursão.

#### 10.2.1. Exemplo 1 - Função não recursiva.

```
int fatorial (int n)
{
    int t, resp;
    resp = 1;
    for (t=1; t <= n; t++) resp = resp * t;
    return (resp);
}
```

Neste exemplo é passado o valor a ser calculado através da variável "n", à função fatorial(). Esta função executa um loop de "t" vezes, onde a variável "resp" é multiplicada pelas iterações de "t".

**10.2.2. Exemplo 2 - Função recursiva.**

```
int fat_rec (int n)
{
    int resp;
    if (n <= 1) return (1);
    resp = fat_rec(n-1) * n; // chamada recursiva
    return (resp);
}
```

Neste exemplo, quando `fat_rec()` é chamada com argumento menor ou igual a 1, ela retorna o valor 1 como resposta, caso contrário, ela devolve o produto de `fat_rec(n-1)*n`. Para avaliar esta expressão, `fat_rec()` é chamada com `n-1`; isso acontece até que `n` se iguale a 1 e as chamadas à função começam a retornar.

O uso de funções recursivas permite criar versões mais simples de vários algoritmos.

**10.3. Exercícios**

- 1- Crie um formulário no C builder para calcular o fatorial de um número, com e sem recursividade. Use as funções dos exemplos 2.1 e 2.2. Verifique passo a passo à execução do programa e explique o funcionamento.
- 2- Faça um programa no C builder para calcular  $x^y$  (  $x$  elevado a  $y$ ) usando uma função recursiva.

## 11. EXERCÍCIOS COM VETORES

---

- 1) Fazer um algoritmo para calcular o número de alunos que tiraram nota acima da nota da média da turma. As notas devem ser armazenadas em um vetor de 100 elementos.
  
- 2) Fazer um algoritmo para preencher um vetor de 100 elementos, colocando 0 (zero) nas posições pares do vetor, e 1 (um) nas posições ímpares.
  
- 3) Fazer um algoritmo para ler uma string qualquer fornecida por um usuário, e escrever na ordem inversa da que foi fornecida.
  
- 4) Escreva um algoritmo que classifique um vetor V de 100 elementos (inteiros e positivos), copiando para um vetor A os elementos pares de V e para um Vetor B os elementos ímpares de V.
  
- 5) Escreva um algoritmo para encontrar um elemento X qualquer fornecido pelo usuário, em um vetor V de 100 elementos do tipo inteiro, e para um vetor S de 100 elementos do tipo caracter. Caso seja encontrado o elemento procurado, apresentar a posição do elemento procurado dentro de V.

## 12 -EXERCÍCIOS COM MATRIZES

---

1) Elabore um algoritmo para gerar uma matriz de 100 x 100 elementos e preencher cada campo, conforme a sequência apresentado abaixo:

1	2	4	7	11	...
2	3	5	8	12	...
3	4	6	9	13	...
4	5	7	10	...	...
5	...	...	...	...	...
...	...	...	...	...	...

2) Elabore um algoritmo, que gere valores aleatórios, inteiros e positivos entre 0 e 4, e movimente o elemento "\*" (asterisco) da matriz apresentada abaixo, segundo a seguinte orientação:

valor gerado = 0: retorna o elemento \* para a posição inicial;

valor gerado = 1: move o elemento \* uma linha acima;

valor gerado = 2: move o elemento \* uma linha abaixo;

valor gerado = 3: move o elemento \* uma coluna para a direita;

valor gerado = 4: move o elemento \* uma linha para a esquerda;

O elemento \* não deve sair fora dos limites da matriz. O movimento deve ser executado na matriz associado ao timer, ou ao clique de um botão. As células que não contém o \* devem estar sempre com '0'.

0	0	0	0	0
0	0	0	0	0
0	0	*	0	0
0	0	0	0	0
0	0	0	0	0



3) Escreva um algoritmo em portugol que preencha uma matriz 4 x 4 com 0, e a sua diagonal secundária com o resultado da soma do conteúdo da linha, com o conteúdo da coluna.

4) Escreva o algoritmo do exercício 3, em linguagem C, e modifique o programa para gerar o número aleatório associado ao evento "timer". O valor gerado para o movimento deve ser passado como parâmetro para uma função que executará o movimento do elemento \* na matriz.

5) Um letreiro digital, pode ser representado por uma matriz 5 x 100, como exemplifica a figura abaixo. Escreva um algoritmo para movimentar a letra 'A', da esquerda para a direita até chegar na última coluna.

0	1	0	0	0	0	0	0	0	0	0	0	...	0
1	0	1	0	0	0	0	0	0	0	0	0	...	0
1	1	1	0	0	0	0	0	0	0	0	0	...	0
1	0	1	0	0	0	0	0	0	0	0	0	...	0
1	0	1	0	0	0	0	0	0	0	0	0	...	0

6) Escreva um algoritmo para calcular a soma, a subtração, a divisão e a multiplicação entre duas matrizes A e B, fornecidas pelo usuário.

7) Escreva um algoritmo ou programa em C, para colocar em ordem alfabética as strings armazenadas na matriz abaixo:

B	a	n	A	n	a	\0	
M	a	ç	A	\0			
A	b	a	C	a	t	e	\0
P	e	r	A	\0			
L	a	r	A	n	j	a	\0

## 13. EVENTOS DE FORMULÁRIO E VARIÁVEIS EXTERNAS

---

### 13.1. EXERCÍCIO PROPOSTO

O código de programa apresentado neste exercício calcula a média entre N valores fornecidos por um usuário. O programa usa dois formulários. Um para saber quantos valores o usuário irá digitar, e para apresentar a média. O segundo formulário é usado para ler cada valor fornecido pelo usuário. A quantidade de valores fornecidos para o programa, depende do tamanho do vetor que os armazena.

Pede-se:

1. Digite e execute o programa especificado e responda as seguintes perguntas:

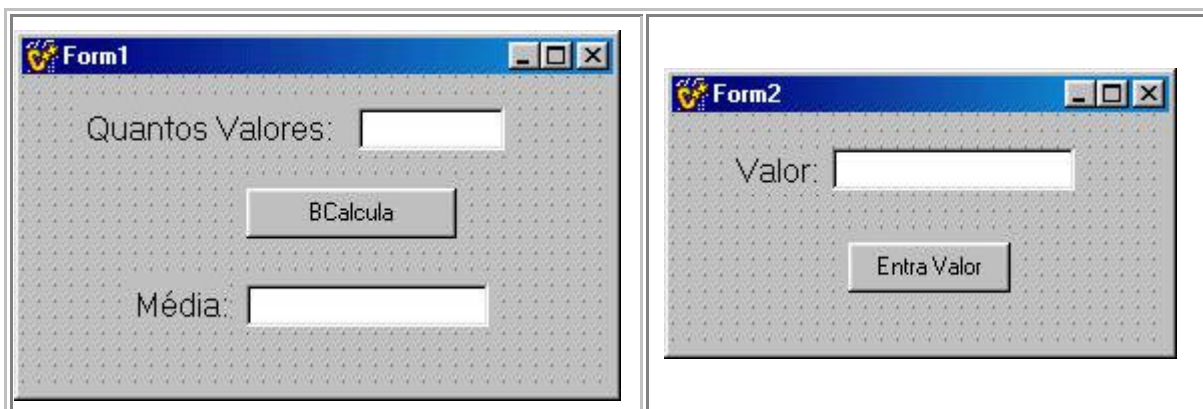
- a) O programa funciona corretamente? Se “sim”, mostre alguns resultados gerados. Se “não”, explique o que está errado.
- b) O que significa ShowModal()?
- c) Que alteração deve ser feita no programa para que o valor da média seja apresentado como uma string? (Mostre as alterações necessárias).
- d) Apresente pelo menos 3 formatos para o comando printf(), e os resultados obtidos na execução do programa.

2. Que alterações podem ser feitas (se for possível), para que o programa calcule a média para N valores fornecidos por um usuário, sem perguntar previamente o valor de N.

**Obs.:** Para todas as modificações feitas no programa, transcreva para a folha do exercício, o trecho modificado correspondente.

**Dica:** A equipe deve criar uma nova aplicação (botões, caixas de texto, etc), e inserir nos devidos locais o código correspondente, caso contrário o programa pode não compilar.

## 13.2. LISTAGEM DO PROGRAMA



Esta aplicação possui dois formulários, portanto duas Units serão criadas para o mesmo Project.

### 13.2.1. Umedia1.cpp

```
//-----
#include <vcl.h>
#pragma hdrstop
#include "Umedia1.h"
#include "Umedia.h"
#include <stdlib.h>
//-----
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
int N;      // quantidade de valores a serem fornecidos pelo usuário
int x;      // quantidade de valores que estão sendo "   "   "
extern double soma; // guarda a soma dos valores fornecidos
double media;      // armazena a média dos "   "
//-----
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    x = 0; // quantidade inicial de valores
}
```

```

//-----
void __fastcall TForm1::BCalculaClick(TObject *Sender)
{
N = atoi(Edit1->Text.c_str()); // quantidade de valores a serem lidos
                                // fornecidos pelo usuário
Form2->ShowModal();             // abre 2.o formulário (Form2)
media = soma / N;               // calcula a média entre os valores
fornecidos
Edit2->Text = media;           // apresenta a média calculada
}
//-----

```

### **13.2.2. Umedia.cpp**

```

//-----
#include <vcl.h>
#pragma hdrstop
#include <stdlib.h>
#include "Umedia.h"
//-----
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm2 *Form2;
extern int x,N;           // variáveis externas (definidas no Form1)
double soma;             // guarda somatório dos valores fornecidos
double valores[20];     // armazena valores reais em um vetor
//-----
__fastcall TForm2::TForm2(TComponent* Owner)
    : TForm(Owner)
{
}
//-----
void __fastcall TForm2::BEntraValorClick(TObject *Sender)
{
if (x < N)               // se quantidade de valores fornecidos
{
                        // é menor que a esperada (N), ainda
                        // existe espaço no vetor para armazenar
    valores[x] = atof(Edit1->Text.c_str()); // coloca valor na última
}
}

```

```

// posição livre do vetor
x++; // incrementa qtde. de valores fornecidos
Edit1->Text = ""; // limpa caixa de texto
if (x==N) // se atingir o limite de valores para o vetor

{ // então fecha o Form2 e zera quantidade x
  Close();
  x=0;
}
}
}
//-----
void __fastcall TForm2::FormClose(TObject *Sender, TCloseAction &Action)
{
  for (int y = 0; y < N; y++)
  {
    soma = soma + valores[y]; // ao fechar o Form2 calcula a soma dos
                             // valores do vetor
  }
}
//-----
```

## 14. ROTINAS DE ORDENAÇÃO

---

1 - Dado o vetor V2, o que será apresentado após a execução do trecho de programa abaixo:

V2 =

m	a	r	i	a	\0
---	---	---	---	---	----

```
void ResolveV1(char item[], int count)
{
    int a;
    int troca;
    char t;
    do{
        troca = 0;
        for(a=count-1; a>0; --a)
        {
            if (item[a-1] > item[a])
            {
                t = item[a-1];
                item[a-1] = item[a];
                item[a] = t;
                troca = 1;
            }
        }
        for(a = 1 ; a < count; ++a)
        {
            if (item[a-1] > item[a])
            {
                t = item[a-1];
                item[a-1] = item[a];
                item[a] = t;
                troca = 1;
            }
        }
    }while(troca == 1);
}
```

2- Dado o vetor V1, o que será apresentado após a execução do trecho de programa abaixo:

V1 =

d	b	a	e	f	\0
---	---	---	---	---	----

```
void ResolveV1(char item[], int count)
{
    int a;
    int troca;
    char t;
    do{
        troca = 0;
        for(a=count-1; a>0; --a)
        {
            if (item[a-1] > item[a])
            {
                t = item[a-1];
                item[a-1] = item[a];
                item[a] = t;
                troca = 1;
            }
        }
        for(a = 1 ; a < count; ++a)
        {
            if (item[a-1] > item[a])
            {
                t = item[a-1];
                item[a-1] = item[a];
                item[a] = t;
                troca = 1;
            }
        }
    } while(troca == 1);
}
```

3- Dado o vetor V3, o que será apresentado após a execução do trecho de programa abaixo:

V3 =

m	a	r	t	e	\0
---	---	---	---	---	----

```
void ResolveV3(char item[], int count)
{
    int a, b;
    char t;
    for(a = 1; a < count; ++a)
    {
        t = item[a];
        for (b = a-1; b >= 0 && t < item[b]; b--)
        {
            item[b+1] = item[b];
        }
        item[b+1] = t;
    }
}
```



## 15. COMPONENTES DO C++ BUILDER E SUAS PRINCIPAIS PROPRIEDADES

### 15.1. BITBTN

Cria um botão com uma determinada função e que pode aparecer com uma figura e um texto.



ícone do bitbtn

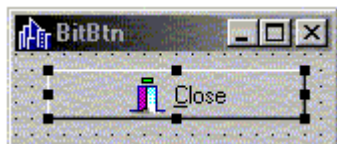
#### 15.1.1. Principais propriedades

a) **Kind:**

- 1) *bkCustom*: selecione a imagem que deve aparecer na propriedade Glyph e selecione um ModalResult para o botão ou uma função no evento OnClick.
- 2) *bkOK*: uma marca verde de checado e um texto “OK” aparecem na face do botão e o valor do ModalResult é mrOK.
- 3) *bkCancel*: Um X vermelho e o texto “Cancel” aparecem na face do botão e o valor do ModalResult é mrCancel.
- 4) *bkYes*: Uma marca verde de checado e um texto “Yes” aparecem na face do botão e o valor do ModalResult é mrYes..
- 5) *bkNo*: Um símbolo vermelho e o texto “No” aparecem na face do botão e o valor do ModalResult é mrNo. B

- 6) *kClose*: Uma porta com o texto “Close” aparece na face do botão e quando o usuário seleciona o botão, o formulário fecha.

### 15.1.2. Exemplo



Nas propriedades, opção Kind, escolher bkClose.

## 15.2 CHECKBOX

É utilizado quando o usuário só pode escolher entre Sim/Não ou Verdadeiro/Falso, sendo assim decisões binárias. Pode-se selecionar mais de um check box em um grupo.



ícone do checkbox

### 15.2.1. Principais propriedades

- a) **Checked:** Pode ser True/False. No caso de verdadeiro, uma marca de checado aparece na caixa, indicando a opção selecionada.
- b) **AllowGrayed:** Determina se o check box pode ter dois (False) ou três estados (True).
- c) **State:** Determina os vários estados que o check box pode apresentar.
  - 1) *cbUnchecked*: não apresenta a marca de selecionado, dizendo que o usuário ainda não selecionou o check box.

- 2) *cbChecked*: apresenta uma marca dizendo que o usuário selecionou o check box.
- 3) *cbGrayed*: indica um estado no qual o check box não está selecionado nem não selecionado, sendo designado sua função dentro do seu programa.

### **15.2.2. Exemplo**



```
void __fastcall TForm1::CheckBox1Click(TObject *Sender)
{
if (CheckBox1->Checked==True)
ShowMessage("Leitura Verificada");
}
```

### **15.3. COMBOBOX**

Mostra uma lista de escolhas combinadas na forma de um list box e um edit box. O usuário pode incluir dados na área do edit box ou selecionar um item no mesmo.



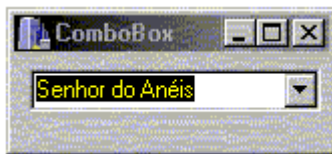
ícone do combobox

#### **15.3.1. Principais propriedades**

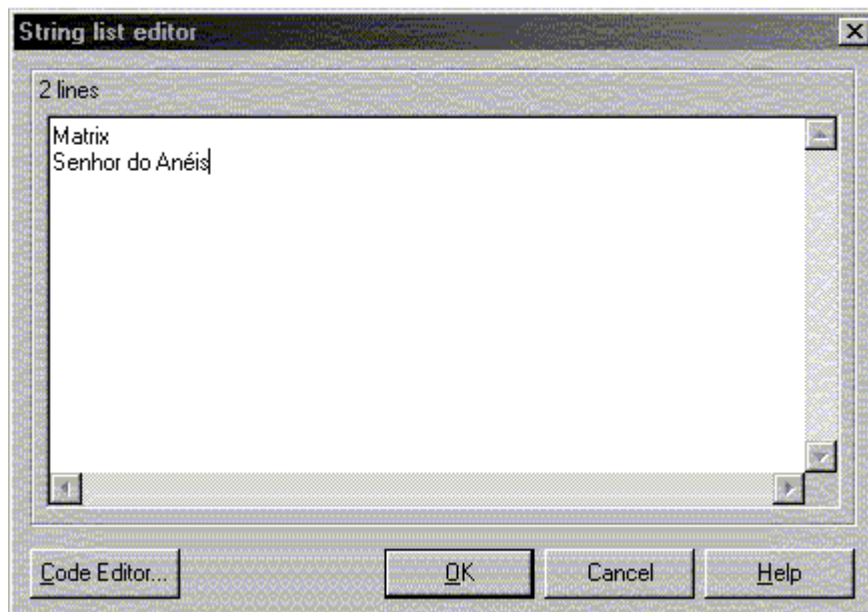
- a) **Style:**

- 1) *csDropDown*: como um list box mas não mostra os dados antes de o usuário apertar o botão.
- 2) *csDropDownList*: permite somente leitura.
- 3) *csSimple*: criar um combo box com uma quantidade fixa de itens.

### 15.3.2. Exemplo



Nas propriedades, opção Items, adicionar os nomes “Matrix” e “Senhor dos Anéis”.  
Na opção Text, escrever “Matrix”.



```
void __fastcall TForm1::ComboBox1Click(TObject *Sender)
{
if(ComboBox1->Text=="Matrix")
ShowMessage("Você escolheu o filme Matrix");
```

```
else ShowMessage("Você escolheu o filme Senhor dos Anéis");  
}
```

## 15.4. LISTBOX

Mostra uma lista de escolhas onde o usuário pode selecionar uma delas.

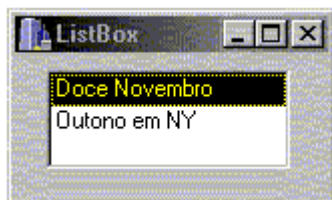


ícone do listbox

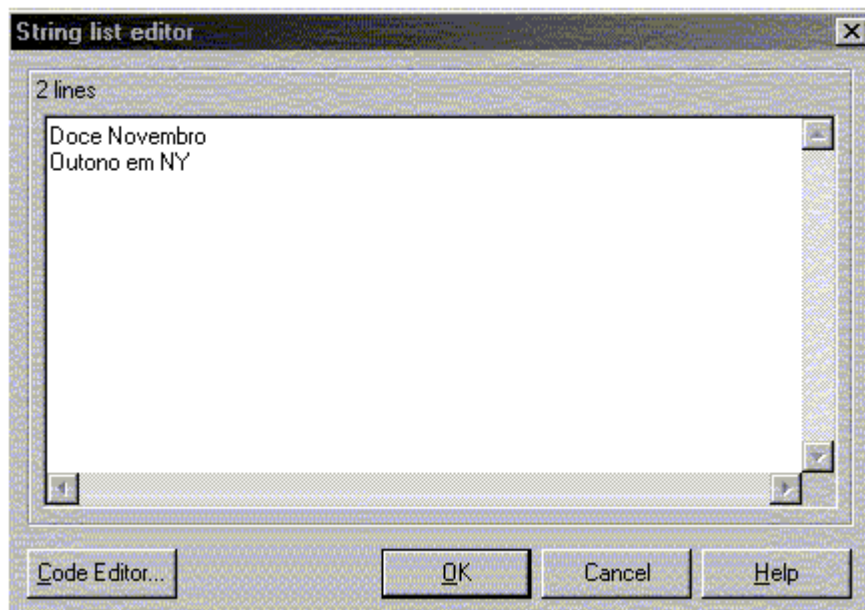
### 15.4.1. Principais propriedades

- a) **MultiSelect:** caso True, o usuário pode selecionar mais de um item.
- b) **Style:** determina como o list box mostra seus itens.

### 15.4.2. Exemplo



Nas propriedades, opção Items, adicionar os nomes “Doce Novembro” e “Outono em NY”.



```
void __fastcall TForm1::ListBox1Click(TObject *Sender)
{
if (ListBox1->ItemIndex==0)
ShowMessage("Você escolheu o filme Doce Novembro");
if (ListBox1->ItemIndex==1)
ShowMessage("Você escolheu o filme Outono em NY");
}
```

### **15.5. PAGECONTROL**

Uma página capaz de criar várias caixas de diálogo. Usado para definir seções de informações dentro da mesma janela. O usuário pode trocar de página clicando na tira da página que aparece no topo da janela. Clicando com o botão direito do mouse, pode-se criar uma nova página clicando na opção New Page.

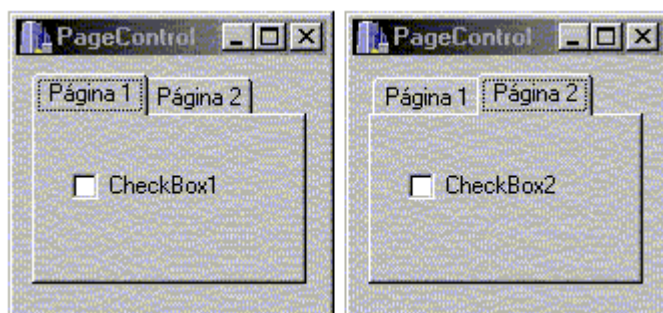


ícone do pagecontrol

### 15.5.1. Principais comandos

- a) **MultiLine:** se False, quando o número de tiras de páginas for maior do que o número de tiras que cabem no topo do botão, elas estarão apresentadas em uma única linha e o usuário terá que clicar na seta de rolagem para ter acesso às demais tiras, caso True, as tiras estarão dispostas em várias linhas.
- b) **ActivePage:** determina qual página é selecionada pelo usuário.
- c) **Style:** determina a aparência das tiras.

### 15.5.2. Exemplo



Clique com o botão direito em cima do mouse e selecione a opção New Page.

```
void __fastcall TForm1::CheckBox1Click(TObject *Sender)
{
if(PageControl1->ActivePage==TabSheet1)
ShowMessage("Você está na página 1");
}
//-----
void __fastcall TForm1::CheckBox2Click(TObject *Sender)
{
if(PageControl1->ActivePage==TabSheet2)
```

```
ShowMessage("Você está na página 2");
}
```

## **15.6. RADIOBUTTON**

Sua função é a mesma do CheckBox, com um diferencial de que só se pode selecionar um radio button em um grupo.



ícone do radiobutton

### **15.6.1. Principais propriedades**

- a) **Caption:** seleciona um valor para o radio button
- b) **Checked:** quando selecionado, aparece um círculo preto dentro do radio button indicando seu estado.

### **15.6.2. Exemplo**



```
void __fastcall TForm1::RadioButton1Click(TObject *Sender)
{
if (RadioButton1->Checked==True)
ShowMessage("Leitura 1 Verificada");
}
//-----
void __fastcall TForm1::RadioButton2Click(TObject *Sender)
```



```
{  
if (RadioButton2->Checked==True)  
ShowMessage("Leitura 2 Verificada");  
}
```

## 15.7. RADIOGROUP

Cria uma caixa de grupo que contem somente radio buttons. Quando o usuário seleciona um radio button, automaticamente os outros radio buttons ficam não selecionados.



ícone do radiogroup

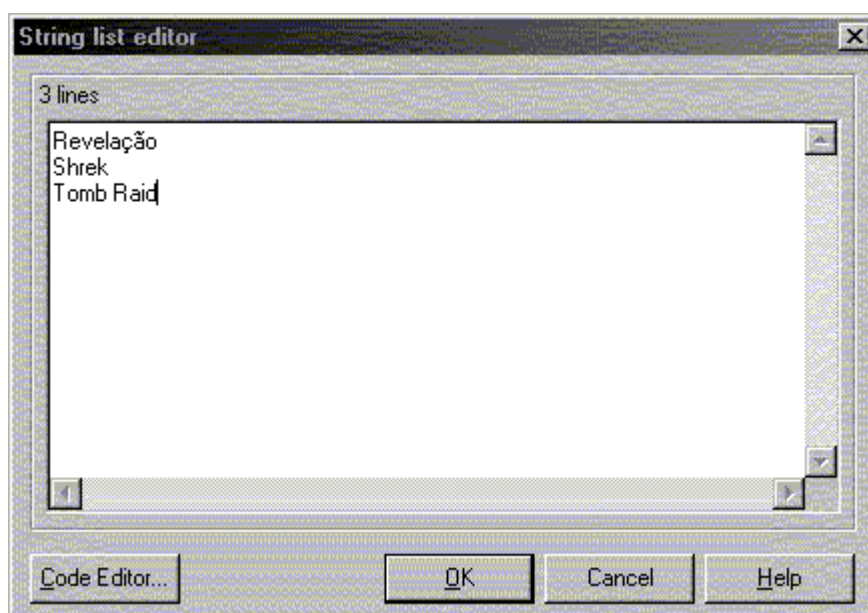
### 15.7.1. Principais propriedades

- a) **Items:** nesta propriedade pode-se inserir os radio buttons que apareceram dentro do RadioGroup.
- b) **ItemIndex:** determina o valor que o radio button recebe quando selecionado.
- c) **Columns:** mostra os radio buttons em uma coluna ou mais

### 15.7.2. Exemplo



Nas propriedades, opção Items, adicionar os nomes Revelação, Shrek e Tomb Raid.



```
void __fastcall TForm1::RadioGroup1Click(TObject *Sender)
{
if(RadioGroup1->ItemIndex==0)
ShowMessage("Você escolheu o filme Revelação");
if(RadioGroup1->ItemIndex==1)
ShowMessage("Você escolheu o filme Shrek");
if(RadioGroup1->ItemIndex==2)
ShowMessage("Você escolheu o filme Tomb Raid");
}
```

### **15.8. SCROLLBAR**

Oferece uma maneira de mudar a área visual de um formulário ou de uma lista. Também se pode mudar a taxa de valores de um incremento.

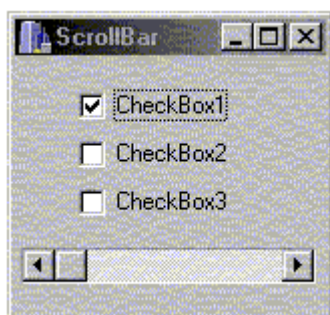


ícone do scrollbar

### 15.8.1. Principais propriedades

- a) **LargeChange:** determina a distância que a barra varia quando se clica em qualquer lado da barra.
- b) **Min and Max:** determina quantas posições são possíveis serem movidas.
- c) **Position:** permite calcular a distância movida da barra ou indicar sua posição.

### 15.8.2. Exemplo



Em propriedades, na opção Max, digite 2.

```
void __fastcall TForm1::ScrollBar1Change(TObject *Sender)
{
if(ScrollBar1->Position==0)
CheckBox1->Checked=true;
else CheckBox1->Checked=false;
if(ScrollBar1->Position==1)
CheckBox2->Checked=true;
else CheckBox2->Checked=false;
if(ScrollBar1->Position==2)
CheckBox3->Checked=true;
```

```
else CheckBox3->Checked=false;  
}
```

### **15.9. SPEEDBUTTON**

Proporciona que apareçam diferentes imagens para diferentes estados do botão como selecionado, não selecionado e desativado. Speed buttons podem ser agrupados com um panel para criar uma barra de ferramentas.

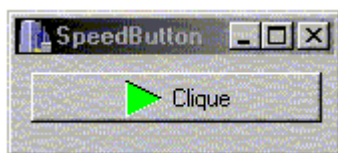


ícone do speedbutton

#### **15.9.1. Principais Propriedades**

- a) **Glyph:** seleciona o arquivo que contem a figura que aparecerá no botão.
- b) **NumGlyph:** oferece até quatro imagens da figura que está no Glyph, mas estas devem ser do mesmo tamanho e estar em colunas.

#### **15.9.2. Exemplo**



Em propriedades, clique na opção Glyph, escolha um bitmap e na opção Caption, digite um texto (se desejar). Para seu SpeedButton estar completo, basta atribuir-lhe uma tarefa na propriedade Events.

## **15.10. STRINGGRID**

Cria uma tabela em que você pode mostrar dados em colunas e linhas. Ele possui muitas propriedades para controlar a aparência da tabela, assim como muitos eventos.

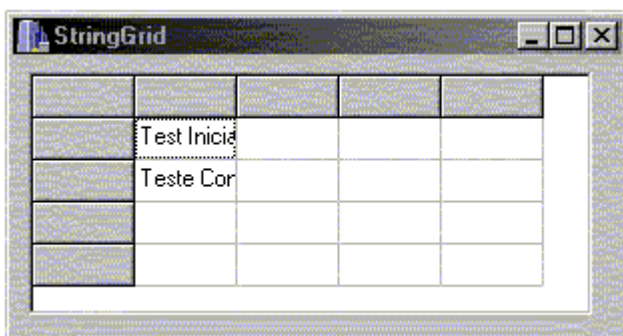


ícone do stringgrid

### **15.10.1. Principais Propriedades**

- a) **ColCount:** exibe o número de colunas que a tabela possui e permite aumentar ou diminuir, sempre em relação ao lado direito da mesma.
- b) **FixedCols:** altera as propriedades das colunas a partir da esquerda e é utilizado especialmente para diferenciar estas células das demais, para a criação de títulos, entre outros.
- c) **FixedRows:** altera as propriedades das linhas a partir do lado superior e é utilizado especialmente para diferenciar estas células das demais, para a criação de títulos, entre outros.
- d) **RowCount:** exibe o número de linhas que a tabela possui e permite aumentar ou diminuir, sempre em relação ao lado inferior da mesma.
- e) **Cells:** são as chamadas células da tabela e são localizadas como em matrizes.

### 15.10.2. Exemplo



```
void __fastcall TForm1::StringGrid1Click(TObject *Sender)
{
StringGrid1->Cells[1][1]="Test Iniciado";
if(StringGrid1->Cells[1][1]=="Test Iniciado")StringGrid1->Cells[1][2]="Teste Concluído";
}
```

### 15.11. TABCONTROL

Ao contrário do Page Control, o Tab Control é um objeto simples que contém várias páginas com comando iguais.



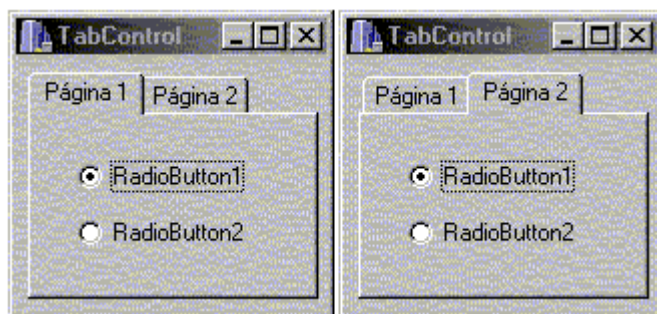
ícone do tabcontrol

#### 15.11.1. Principais Propriedades

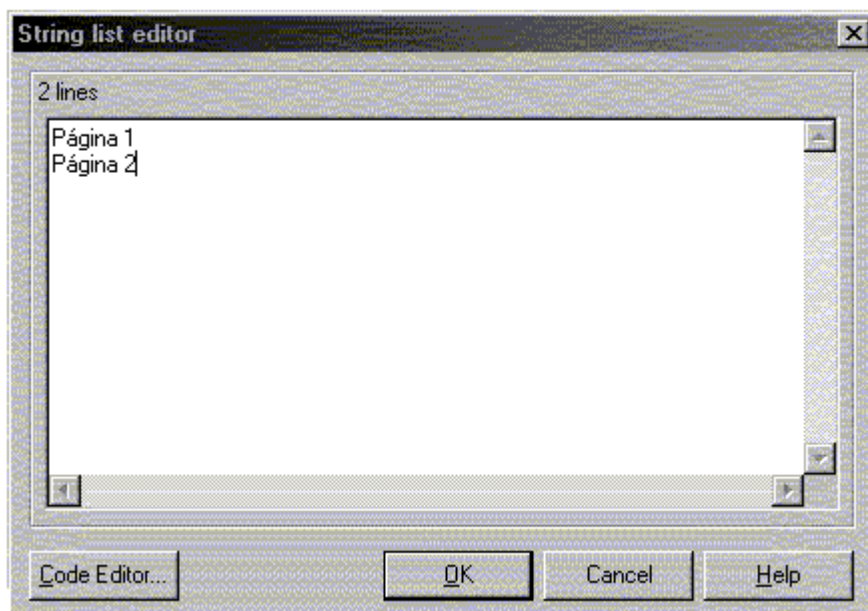
- a) **Tabs:** clicando aqui, pode-se criar novas páginas, basta adicionar o nome das mesmas.
- b) **TabIndex:** alterando entre 0 e o número de páginas criadas, pode-se acessar cada uma delas.

- c) **MultiLine:** se False, quando o número de tiras de páginas for maior do que o número de tiras que cabem no topo do botão, elas estarão apresentadas em uma única linha e o usuário terá que clicar na seta de rolagem para ter acesso às demais tiras, caso True, as tiras estarão dispostas em várias linhas.

### 15.11.2. Exemplo



Em propriedades, na opção Tabs, digitar Página 1 e Página 2.



```
void __fastcall TForm1::RadioButton5Click(TObject *Sender)
{
if(TabControl1->TabIndex==0)
```

```
ShowMessage("Check Box 1, Página 1");
if(TabControl1->TabIndex==1)
ShowMessage("Check Box 1, Página 2");
}
//-----
void __fastcall TForm1::RadioButton6Click(TObject *Sender)
{
if(TabControl1->TabIndex==0)
ShowMessage("Check Box 2, Página 1");
if(TabControl1->TabIndex==1)
ShowMessage("Check Box 2, Página 2");
}
```